

# 第六章 并发程序设计

## 目录

<b>1 并发程序设计</b>	<b>1</b>
1.1 并发程序设计的概念 . . . . .	1
1.1.1 顺序程序设计 . . . . .	1
1.1.2 并发程序设计 . . . . .	1
1.2 并发进程的制约关系 . . . . .	2
1.2.1 无关与交互的并发进程 . . . . .	2
1.2.2 与时间相关的程序设计错误 . . . . .	3
1.2.2.1 结果错误 . . . . .	3
1.2.2.2 永远等待 . . . . .	4
1.2.3 进程执行的制约关系 . . . . .	4
1.2.3.1 进程互斥（竞争） . . . . .	5
1.2.3.2 进程同步（协作） . . . . .	5
1.3 操作系统并发问题解决方案 . . . . .	5
<b>2 临界区管理</b>	<b>6</b>
2.1 临界区的概念 . . . . .	6
2.2 临界区管理实现的尝试 . . . . .	7
2.3 临界区管理的硬件方式 . . . . .	8
2.3.1 关中断 . . . . .	8
2.3.2 测试并设置 (TS) 指令 . . . . .	8
2.3.3 对换指令 . . . . .	8
<b>3 PV 操作</b>	<b>8</b>
3.1 PV 操作与进程互斥 . . . . .	8
3.1.1 PV 操作 . . . . .	8
3.1.2 PV 操作解决进程互斥问题 . . . . .	11
3.1.2.1 飞机售票问题 . . . . .	11
3.1.2.2 哲学家就餐问题 . . . . .	12
3.2 PV 操作与进程同步 . . . . .	14
3.2.1 PV 操作解决进程同步问题的基本思路 . . . . .	14
3.2.2 生产者消费者问题 . . . . .	14
3.2.2.1 生产者消费者问题分析 . . . . .	15
3.2.2.2 一个生产者、一个消费者共享一个缓冲区问题 . . . . .	15
3.2.2.3 一个生产者、一个消费者共享多个缓冲区问题 . . . . .	15
3.2.2.4 多个生产者、多个消费者共享多个缓冲区问题 . . . . .	16
3.2.3 苹果-橘子问题 . . . . .	16
<b>4 信号量与 PV 操作习题</b>	<b>17</b>
4.1 读者/写者问题 . . . . .	17
4.1.1 读者优先 . . . . .	17
4.1.2 写者优先 . . . . .	18
4.1.3 读写平衡 . . . . .	19
4.2 睡眠的理发师问题 . . . . .	19

4.3 农夫猎人问题 . . . . .	20
4.4 银行业务问题 . . . . .	20
4.5 缓冲区管理 . . . . .	21
4.6 售票问题 . . . . .	21
4.7 吸烟者问题 . . . . .	22
4.8 独木桥问题 . . . . .	22
4.8.1 独木桥问题 1 . . . . .	22
4.8.2 独木桥问题 2 . . . . .	22
4.8.3 独木桥问题 3 . . . . .	23
4.8.4 独木桥问题 4 . . . . .	23
<b>5 管程</b> . . . . .	<b>24</b>
5.1 管程概述 . . . . .	24
5.1.1 管程的提出 . . . . .	24
5.1.2 管程的规格定义与实现思路 . . . . .	24
5.2 霍尔管程 . . . . .	26
5.2.1 霍尔管程的数据结构 . . . . .	26
5.2.2 霍尔管程的 enter() 和 leave() 操作 . . . . .	27
5.2.3 霍尔管程的 wait() 操作 . . . . .	27
5.2.4 霍尔管程的 signal() 操作 . . . . .	27
5.3 霍尔管程求解进程互斥与同步问题 . . . . .	27
5.3.1 霍尔管程求解读者/写者问题 . . . . .	27
5.3.2 霍尔管程求解哲学家就餐问题 . . . . .	28
5.3.3 霍尔管程求解生产者消费者问题 . . . . .	28
5.3.4 霍尔管程求解苹果桔子问题 . . . . .	29
<b>6 进程通信</b> . . . . .	<b>30</b>
6.1 进程通信概述 . . . . .	30
6.1.1 进程直接通信 . . . . .	30
6.1.2 进程间接通信 . . . . .	30
6.1.3 消息缓冲通信 . . . . .	32
6.1.4 管道和套接字 . . . . .	33
6.2 高级进程通信机制 . . . . .	33
6.2.1 基于流的进程通信 . . . . .	33
6.2.2 基于 RPC 的高级通信规约 . . . . .	34
<b>7 死锁</b> . . . . .	<b>35</b>
7.1 死锁的产生 . . . . .	35
7.1.1 死锁的定义 . . . . .	35
7.1.2 死锁的产生 . . . . .	35
7.1.2.1 进程推进顺序不当产生死锁 . . . . .	35
7.1.2.2 PV 操作使用不当产生死锁 . . . . .	36
7.1.2.3 资源分配不当引起死锁 . . . . .	36
7.1.2.4 对临时性资源使用不加限制引起死锁 . . . . .	36
7.2 死锁的防止 . . . . .	36
7.2.1 死锁产生的条件 . . . . .	36
7.2.2 死锁的防止 . . . . .	36
7.2.2.1 静态分配策略 . . . . .	37

7.2.2.2 层次分配策略 . . . . .	37
7.3 死锁的避免 . . . . .	37
7.3.1 银行家算法的数据结构 . . . . .	37
7.3.2 银行家算法的思想与实现 . . . . .	38
7.4 死锁的检测 . . . . .	41
7.4.1 利用等待资源表和占用资源表进行死锁检测 . . . . .	42
7.4.2 进程-资源分配图与死锁的检测 . . . . .	43
7.4.3 死锁的解除 . . . . .	44

# 1 并发程序设计

## 1.1 并发程序设计的概念

### 1.1.1 顺序程序设计

程序是实现算法的操作（指令）序列

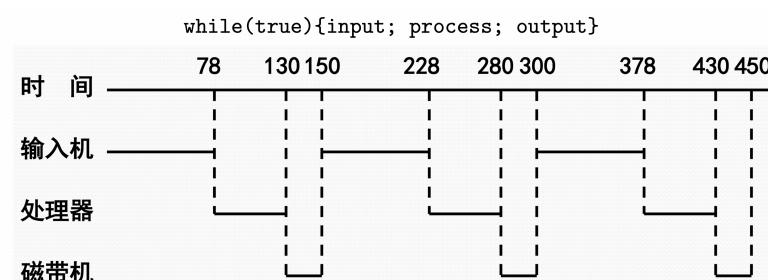
- 程序顺序执行是指其在处理器上的执行是严格有序的，即只有在前一个操作结束后，才能开始后续操作，这称为程序执行的内部顺序性
- 将一个具体问题的求解过程设计成一个程序或者若干个严格按序执行的程序序列，这称为程序执行的外部顺序性
- 传统程序设计方法是基于顺序程序设计的，即将程序设计成顺序执行的指令序列，不同程序也按先后顺序执行

顺序程序设计的特性：

- 程序执行的顺序性：程序指令执行是严格按序的
- 计算环境的封闭性：程序运行时如同独占受操作系统保护的资源
- 计算结果的确定性：程序执行结果与执行速度和执行时段无关
- 计算过程的可再现性：程序对相同数据集的执行轨迹是确定的

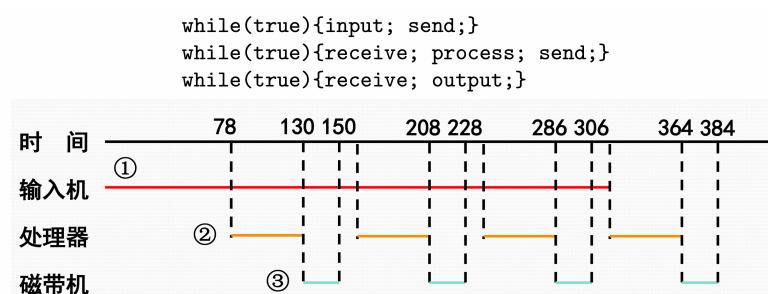
### 1.1.2 并发程序设计

顺序程序设计系统的效率相当低，对于下图例子而言，其处理器的使用效率仅为  $52/(78+52+20) = 35\%$

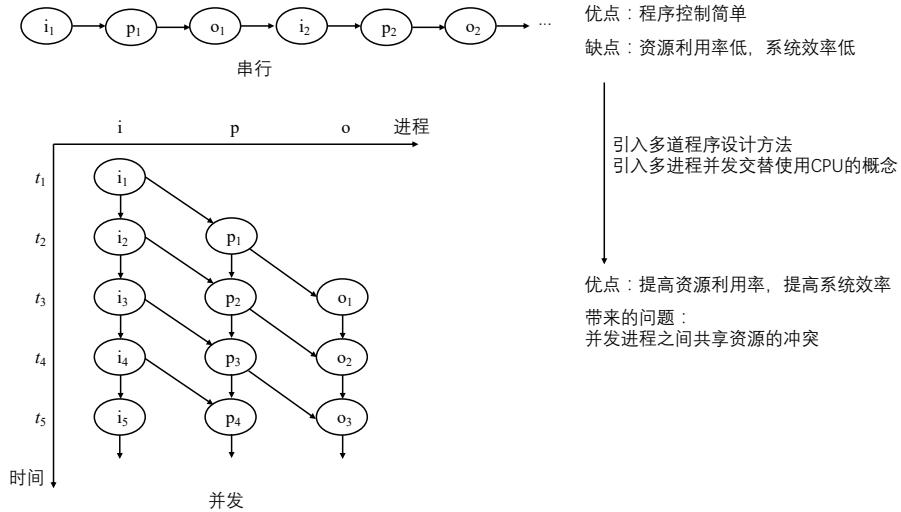


换一种设计思路，设计 3 个独立运行的程序，让它们同时进入多道程序系统去并发执行

- 假定循环次数为  $n$ ，处理器的使用效率  $= (52n)/(78n + 52 + 20)$ ，当  $n \rightarrow \infty$  时，处理器的使用效率约为 67%



## 顺序程序设计与并发程序设计



程序并发执行是指一组程序的执行在时间上是重叠的，所谓重叠的是指一个程序执行第一条指令是在另一个程序执行完最后一条指令之前开始的

- 在宏观上，并发性反应了一个时间段内有几个程序都处于运行但运行尚未结束的状态
- 在微观上，任一时刻都只有一个程序在运行
- 并发实质上是处理器在几个程序之间的多路复用，对有限物力资源强制性使多用户共享，消除计算机部件之间的互等现象，提高资源利用率
- 并发使得程序失去了封闭性、顺序性、确定性和可再现性

并发程序设计的特性：

- **并行性：**多个进程在多道程序系统中并发执行或者在多处理器系统中并行执行，提高了计算效率
- **共享性：**多个进程共享软件资源
- **交互性：**多个进程并发执行时存在制约，增加了程序设计的难度

并发程序设计的优点：

- 对于单处理器系统，能够有效利用资源，让处理器和设备、设备和设备同时工作，充分发挥硬件设备的并行工作能力
- 对于多处理器系统，能够让进程在不同处理器上物理地并行工作，加快计算速度
- 对于简化程序设计而言，编写并发执行的小程序进度较快，更容易保证正确性

## 1.2 并发进程的制约关系

### 1.2.1 无关与交互的并发进程

并发进程可能是无关的，也可能是交互的

- 无关的并发进程：一组并发进程分别在不同的变量集合上运行，一个进程的执行与其他并发进程的进展无关
- 交往的并发进程：一组并发进程共享某些变量，一个进程的执行可能影响其他并发进程的结果

并发进程的无关性是进程的执行与时间无关的一个充分条件，该条件在 1966 年由 Bernstein 提出。假设：

$R(P_i) = a_1, a_2, \dots, a_n$  表示程序  $P_i$  在执行期间所引用的变量集

$W(P_i) = b_1, b_2, \dots, b_m$  表示程序  $P_i$  在执行期间所改变的变量集

若两个进程的程序  $P_1$  和  $P_2$  能满足 Bernstein 条件，即引用变量集与改变变量集的交集为空集：

$$R(P_1) \cap W(P_2) \cup R(P_2) \cap W(P_1) \cup W(P_1) \cap W(P_2) = \emptyset$$

则并发进程的执行与时间无关，并发执行的程序可以保持封闭性和可再现性。

- $R(P_1) \cap W(P_2) \cup R(P_2) \cap W(P_1)$ , 表明一个程序在两次读操作之间存储单元的数据不会被改变
- $W(P_1) \cap W(P_2)$ , 表明程序的写操作结果不会丢失

例：有如下四条语句

$$S_1 : a = x + y; S_2 : b = z + 1; S_3 : c = a - b; S_4 : w = c + 1$$

于是有： $R(S_1) = \{x, y\}$ ,  $R(S_2) = \{z\}$ ,  $R(S_3) = \{a, b\}$ ,  $R(S_4) = \{c\}$ ;  $W(S_1) = \{a\}$ ,  $W(S_2) = \{b\}$ ,  $W(S_3) = \{c\}$ ,  $W(S_4) = \{w\}$

- 因为  $S_1$  和  $S_2$  满足 Bernstein 条件，所以它们可以并发执行
- 而其他语句之间因变量交集之并不为空，表明它们之间是存在交互的，并发执行可能会产生与时间有关的错误

### 1.2.2 与时间相关的程序设计错误

两个交互的并发进程，其中一个进程对另一个进程的影响往往是不可预期的，甚至是无法再现的

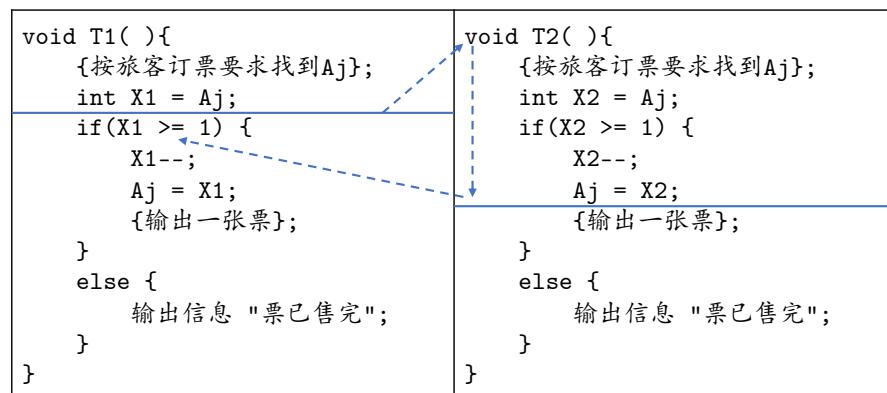
- 这是因为它们执行时的相对速度不可预测

由于一个进程的执行速度通常无法为另一个进程所预知，对于共享公共变量（资源）的并发进程而言，计算结果往往取决于这一组并发进程执行的相对速度，继而可能导致出现错误，这种错误称为与时间有关的错误

- 表现一：结果错误
- 表现二：永远等待

#### 1.2.2.1 结果错误

##### 飞机售票问题



可能出现如下调度情况

```

1 T1: X1 = A[j];           /* X1 = m(m>0) */
2 T2: X2 = A[j];           /* X2 = m */
3 T2: X2--; A[j] = X2; {输出一张票}; /* A[j] = m - 1 */
4 T1: X1--; A[j] = X1; {输出一张票}; /* A[j] = m - 1 */

```

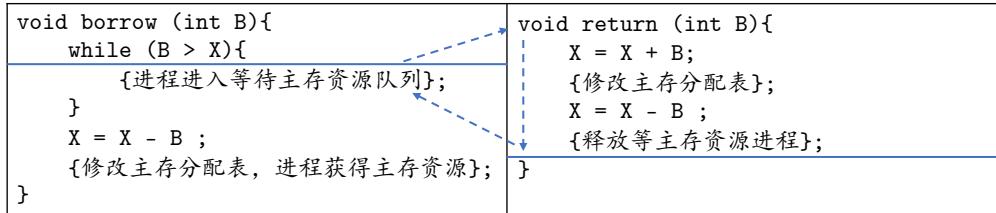
此时出现把同一张票卖给两个旅客的情况

- 两个旅客可能各自都买到一张同天同次航班的机票，可是 $A[j]$ 的值实际上只减去 1，造成余票数不正确
- 特别是，当某次航班只有一张余票时，可能把一张票同时售给两位旅客

### 1.2.2.2 永远等待

申请和归还主存资源问题

```
int x = memory; //memory为初始主存容量
```

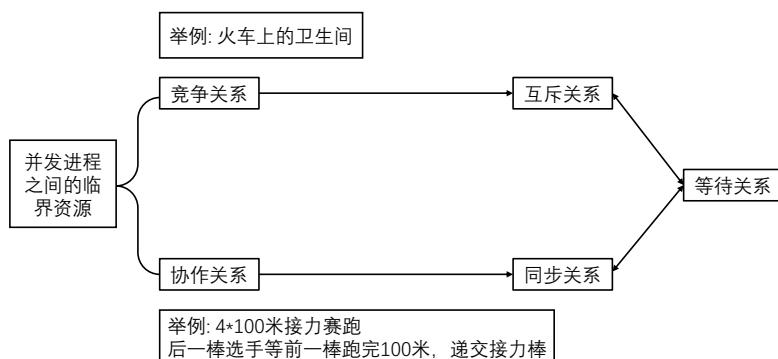


由于 borrow 和 return 共享代表主存物理资源的临界变量 X，对并发执行不加限制会导致错误。例如：

- 进程 P 调用 borrow() 申请主存资源，在执行比较 B 和 X 的大小的指令后，发现  $B > X$  条件成立，但在执行进程进入等待主存资源队列之前被抢占
- 进程 Q 调用 return() 抢先执行，归还所借全部主存资源
- 这时，由于进程 P 尚未成为等待者，return() 中的释放等待主存资源进程相当于空操作，当调用 borrow() 的应用进程再次获得处理器时，因此前判断  $B > X$  条件成立，将正式进入等待主存资源队列，假设后续没有其他进程再来归还主存资源，那么，已经进入等待主存资源队列的进程将处于永远等待的状态

### 1.2.3 进程执行的制约关系

交互的并发进程在执行时的制约关系包括两类：进程互斥与进程同步。只有有效解决进程互斥与进程同步这两种制约关系，才能保证并发进程在交互的执行过程中能够得到合理的结果。



### 1.2.3.1 进程互斥（竞争）

进程互斥是指并发进程之间因相互争夺独占性资源而产生的竞争制约关系

- 一个进程的执行可能影响到同其竞争资源的其他进程，如果两个进程要访问同一资源，那么，一个进程通过操作系统分配得到该资源，另一个将不得不等待
- 资源竞争带来的两个控制问题：
  - 一个是死锁问题：一组进程如果都获得了部分资源，还想要得到其他进程所占有的资源，最终所有的进程将陷入死锁



- 一个是饥饿问题：一个进程由于其他进程总是优先于它而被无限期拖延
- 操作系统需要保证诸进程能互斥地访问临界资源，既要解决饥饿问题，又要解决死锁问题

### 1.2.3.2 进程同步（协作）

进程同步是指并发进程之间为完成共同任务基于某个条件来协调执行先后关系而产生的协作制约关系

- 两个及以上进程基于某个条件来协调它们的活动。一个进程的执行依赖于另一个协作进程的消息或信号，当一个进程没有得到来自于另一个进程的消息或信号时则需等待，直到消息或信号到达才被唤醒
- 进程互斥关系是一种特殊的进程同步关系，即逐次使用互斥共享资源，是对进程使用资源次序上的一种协调

## 1.3 操作系统并发问题解决方案

原语类型	采用策略	同步机制	适用场合	方向
高级通信原语	采用消息传递、共享内存、共享文件策略	消息队列、共享内存、管道通信	解决并发进程通信、同步和互斥问题，适用于面向语句的高级程序设计	
低级通信原语	采用阻塞/唤醒 + 集中临界区（1次测试）策略	管程	解决并发进程同步和互斥问题，不能传递消息，适用于面向语句的高级程序设计	↑ ↓
	采用阻塞/唤醒 + 分散临界区（1次测试）策略	信号量和 PV 操作	解决并发进程同步和互斥问题，不能传递消息，适用于面向指令的低级程序设计	↑ ↓
	采用忙式等待（反复测试）策略	关中断、对换、测试并建立、peterson 算法、dekker 算法	解决并发进程互斥问题，不能传递消息，适用于面向指令的低级程序设计	

## 2 临界区管理

## 2.1 临界区的概念

并发进程中与共享变量有关的程序段叫“临界区”，共享变量代表的资源叫“临界资源”

- 与同一变量有关的临界区分散在各进程的程序段中，而各进程的执行速度不可预见
  - 如果保证进程在临界区执行时，不让另一个进程进入临界区，即各进程对共享变量的访问是互斥的，就不会造成与时间有关的错误

关于临界区的概念是由 Dijkstra 在 1965 年首先提出的。临界区在程序设计中的形式化定义如下，用 `shared` 表示共享变量

```
1 shared <variable>
2 region <variable> do <statement_list>
```

为了正确而有效地使用临界资源，共享变量的并发进程应遵守临界区调度的三个原则

1. 一次至多一个进程能够进入临界区内执行
  2. 如果已有进程在临界区，其他试图进入的进程应等待
  3. 进入临界区内的进程应在有限时间内退出，以便让等待进程中的一个进入

```
/* Process 1 */
void P1(){
    while(true){
        /* preceding code */
        entercritical(Ra);
        /* critical section */
        exitcritical(Ra);
        /* following code */
    }
}

/* Process 2 */
void P2(){
    while(true){
        /* preceding code */
        entercritical(Ra);
        /* critical section */
        exitcritical(Ra);
        /* following code */
    }
}

...
/* Process n */
void Pn(){
    while(true){
        /* preceding code */
        entercritical(Ra);
        /* critical section */
        exitcritical(Ra);
        /* following code */
    }
}
```

临界区是允许嵌套的，其代码结构如下图 (a) 所示。

- 但是随意或者不严谨的嵌套可能导致进程永久地留在其临界区内
  - 如果另有一个进程执行，其代码如图 (b) 所示，这样，当两个进程在差不多的时间进入外层的临界区后，将发现每个进程都被排斥在内层临界区之外，造成无限地等待进入内层临界区，即临界区嵌套使用不当会产生死锁

```
shared x,y;  
region x do{/*外层临界区*/  
    ...  
    假设产生中断  
-----  
    region y do{/*内层临界区*/  
        ...  
    }  
    ...  
}  
}
```

```
shared x,y;  
region y do{/*外层临界区*/  
    ...  
    可能产生死锁  
}  
  
region x do{/*内层临界区*/  
    ...  
    ...  
}
```

(a)

(b)

## 2.2 临界区管理实现的尝试

考虑并发进程在单处理器或共享内存的多处理器计算机系统上的执行情况，这里先讨论采用软件方法实现临界区管理

- 通常假设系统具有存储器访问级的基本互斥性，即同时访问内存中的同一个单元时，必定由存储器进行仲裁使其串行化
- 此外，硬件、操作系统或语言未提供任何支持

临界区管理可采用标志方式，即用标志来表示哪个进程可进入临界区

第一种尝试如下：

```
bool inside1 = false; //P1不在其临界区内
bool inside2 = false; //P2不在其临界区内
cobegin /*cobegin和coend表示括号中的进程是一组并发进程*/
    process P1(){
        while(inside2); //等待 ①
        inside1 = true; ②
        {临界区};
        inside1 = false;
    }
    process P2(){
        while(inside1); //等待 ③
        inside2 = true; ④
        {临界区};
        inside2 = false;
    }
coend
```

如果并发执行轨迹是 ①③②④，则两个进程都满足进入条件，均能进入临界区

第二种尝试是对第一种方案做修正：

```
bool inside1 = false; //P1不在其临界区内
bool inside2 = false; //P2不在其临界区内
cobegin /*cobegin和coend表示括号中的进程是一组并发进程*/
    process P1(){
        inside1 = true; ①
        while(inside2); //等待 ②
        {临界区};
        inside1 = false;
    }
    process P2(){
        inside2 = true; ③
        while(inside1); //等待 ④
        {临界区};
        inside2 = false;
    }
coend
```

如果并发执行轨迹是 ①③②④，则两个进程都被阻塞，均不能进入临界区

解决算法：peterson 算法，但该算法的通用性和效率较差

```
bool inside[2]      //存储进程是否进入临界区
inside[0] = false;
inside[1] = false;
enum{0, 1} turn;   //指示器turn来指示可以由哪个进程进入临界区
cobegin /*cobegin和coend表示括号中的进程是一组并发进程*/
    process P0(){
        inside[0] = true;
        turn = 1;
        while(inside[1] && turn == 1);
        {临界区};
        inside[0] = false;
    }
    process P1(){
        inside[1] = true;
        turn = 0;
        while(inside[0] && turn == 0);
        {临界区};
        inside[1] = false;
    }
coend
```

## 2.3 临界区管理的硬件方式

在单处理器计算机系统中，并发进程不能同时执行，只会交替地执行。为了保证互斥性，仅需保证进程不被中断。

### 2.3.1 关中断

实现互斥最简单的方法是在进程进入临界区时关中断，进程退出临界区时开中断。

- 由于进程上下文切换都是由中断事件引起的，因此关闭中断后进程的执行不会被打断
- 例如，Linux 系统中采用内核函数 cli() (clear interrupt) 和 sti() (set interrupt) 实现关中断与开中断
- 操作系统原语就是基于该思路实现的，但该方式不适合作为通用的互斥机制
  - 临界区的指令长度应短小精悍，否则关中断的时间过长会影响性能和系统效率
  - 它不适用于多处理器系统，一个处理器关中断并不能防止进程在其他处理器上执行相同的临界区代码
  - 若将这项权利赋予用户也存在危险，如果用户未开中断，则系统可能因此而终止

### 2.3.2 测试并设置 (TS) 指令

<pre>//TS指令的处理过程 bool TS(bool &amp;x) {     if(x){         x = false;         return true;     }else         return false; }</pre>	<pre>//TS指令实现进程互斥 bool s = true; cobegin     process Pi( ){ //i=1,2,...,n         while(!TS(s)); //循环请求锁，得到后上锁         {临界区};         s = true; //开锁     } coend</pre>
--	--

### 2.3.3 对换指令

<pre>void SWAP(bool &amp;a, bool &amp;b){     bool temp = a;     a = b;     b = temp; }</pre>	<pre>//对换指令实现进程互斥 bool lock = false; cobegin     process Pi( ){ //i=1,2,...,n         bool keyi = true;         do{             SWAP(keyi,lock);         }while(keyi); //上锁         {临界区};         SWAP(keyi,lock); //开锁     } coend</pre>
---	--

## 3 PV 操作

### 3.1 PV 操作与进程互斥

#### 3.1.1 PV 操作

问题的提出：对于临界区管理的软硬件方法，虽然简单且行之有效，可以正确解决临界区调度问题，但也存在明显缺点：

- 软件算法太复杂，效率低下
- 硬件设施采用忙式等待测试，浪费 CPU 时间
- 将测试能否进入临界区的责任推给各个竞争的进程，削弱了系统的可靠性，加重了用户的编程负担

信号量是一种可动态定义的软件资源，其核心的数据结构就是让操作系统为它建立一个等待进程队列

- 信号量声明：当声明信号量并为其赋予一合理的初值时，相当于声明系统中可用的各类资源，操作系统就为可用的资源建立相关的等待队列
- 申请资源的原语：当申请资源时，如果申请结果是暂时得不到资源（或者是资源不满足），调用申请资源的原语就会将申请进程加入与该资源相关的等待队列中
- 释放资源的原语：当归还资源时，操作系统会检测在该资源的等待队列中有没有等待该资源的进程，如果有，便会将其释放
- 信号量的撤销：注销该信号量指代的资源，撤销与该信号量相关的队列

信号量的一种实现机制——记录型信号量在程序设计中的形式化定义

```

1 typedef struct semaphore{
2     int value;           /* 信号量值 */
3     struct pcb * list; /* 信号量等待进程队列指针 */
4 }
```

- value 为整型变量，系统初始化时为其赋值
  - 正值表示资源可复用次数
  - 0 值表示无资源且无进程等待
  - 负值表示等待队列中进程个数
- list 是等待使用此类资源的进程队列的头指针，初始状态为空队列

设 s 为一个记录型信号量，关于 s 的 PV 操作原语描述如下：

- P(s)：将信号量 s 的 value 值减 1
  - 若结果小于 0，则执行 P 操作的进程被阻塞，进入信号量 s 的 list 所指队列中
  - 若结果大于等于 0，则执行操作的进程继续执行
- V(s)：将信号量 s 的 value 值加 1
  - 若结果不大于 0，则执行 V 操作的进程从信号量 s 的 list 所指队列中释放一个进程，使其转换为就绪态，自己则继续执行
  - 若结果大于 0，则执行 V 操作的进程继续执行

将 PV 操作定义为如下不可中断过程

```

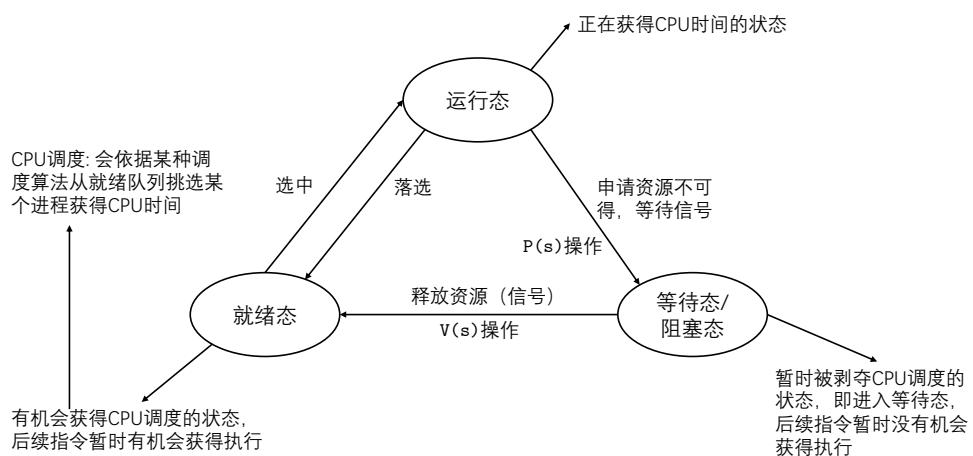
1 // P操作原语
2 void P(semaphore s) {
3     s.value--;           /* 信号量值减1 */
4     if(s.value < 0)
5         sleep(s.list);
6 /**
7 * 若信号量值小于0，执行P操作的进程调用sleep(s.list)阻塞自己，被置成等待信号量s状态并移入s
8 * 信号量队列，转向进程调度程序
9 */
```

```

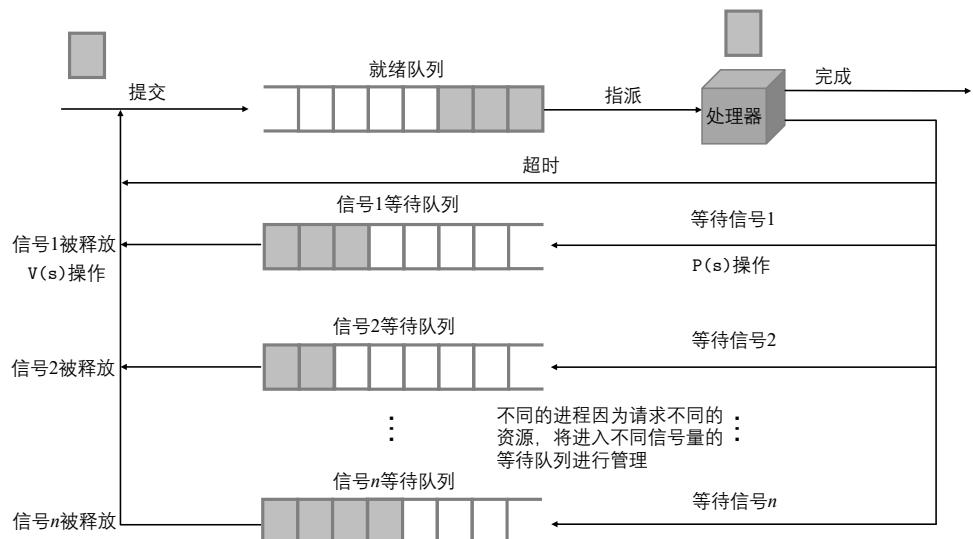
1 // V操作原语
2 void V(semaphore s) {
3     s.value++;           /* 信号量值加1 */
4     if(s.value <= 0)
5         wakeup(s.list);
6     /*
7      * 若信号量值小于等于0，则调用wakeup(s.list)从信号量s队列中释放一个等待信号量s的进程并转换
8      成就绪态，进程则继续执行
9     */
10 }

```

### 信号量与进程状态转换模型



### 信号量与进程队列模型



### 信号量与 PV 操作的推论

- 若信号量 s 的 value 值为正值，此值等于在封锁进程之前对信号量 s 可施行的 P 操作数，即 s.value 代表实际可用的物理资源数。

- 若信号量  $s$  的 value 值为负值，其绝对值等于登记排列在信号量  $s$  的 list 所指队列中等待进程的个数，即等于对信号量  $s$  实施 P 操作而被封锁并进入信号量  $s$  等待队列的进程数。
- P 操作通常意味着请求一个资源，V 操作意味着释放一个资源。在一定条件下，P 操作代表阻塞进程的操作，而 V 操作代表唤醒被阻塞进程的操作。

### 3.1.2 PV 操作解决进程互斥问题

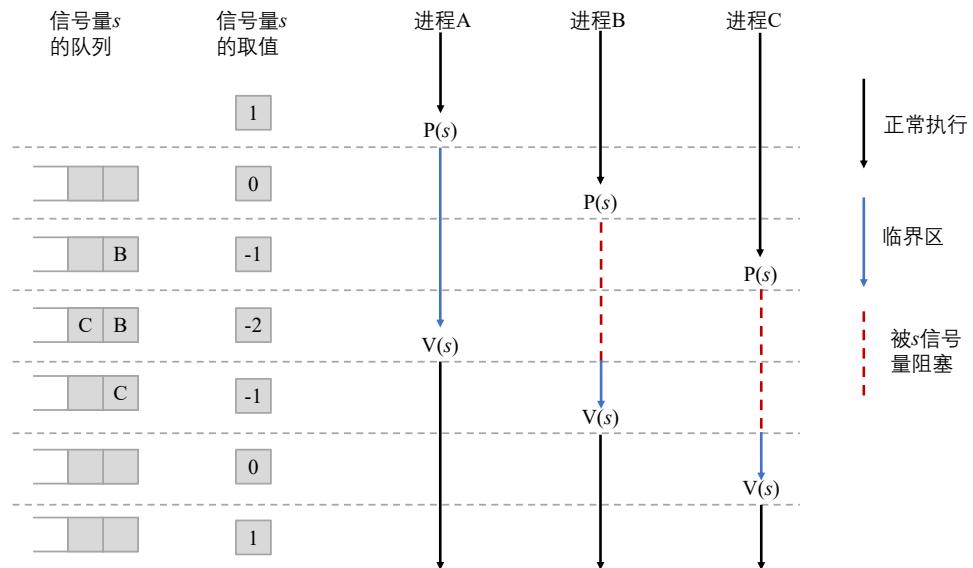
使用信号量与 PV 操作管理并发进程互斥进入临界区的一般形式如下

```

1 semaphore s = 1;
2 cobegin
3   Process Pi(){ /* i=1,⋯⋯,n */
4     ...
5     P(s);      //申请进入临界区
6     {临界区};
7     V(s);      //申请退出临界区
8     ...
9   };
10 coend

```

在表达纯粹互斥关系时信号量初值为 1，且同一个信号量的 P 操作和 V 操作处于同一侧进程之中，但是这种情形不适用于同步关系



#### 3.1.2.1 飞机票售问题

用记录型信号量与 PV 操作解决飞机售票问题

```

1 int A[m];
2 semaphore s;
3 s = 1;
4 cobegin
5   process Pi(){
6     int Xi;
7     L1: 按旅客订票要求找到A[j];

```

```

8   P(s);
9   Xi = A[j];
10  if (Xi >= 1){
11      Xi = Xi - 1;
12      A[j] = Xi;
13      V(s);
14      {输出一张票};
15  }else{
16      V(s);
17      {输出票已售完};
18  }
19  goto L1;
20 }
21 coend

```

对于上述问题而言，只有相同航班的票数才是相关的临界资源，所以用一个信号量处理全部机票的销售过程会影响进程并发度

因此对以上程序做出改进：为每一个航班定义一个信号量，可以定义一个信号量数组，即每一个航班都有其各自的临界区

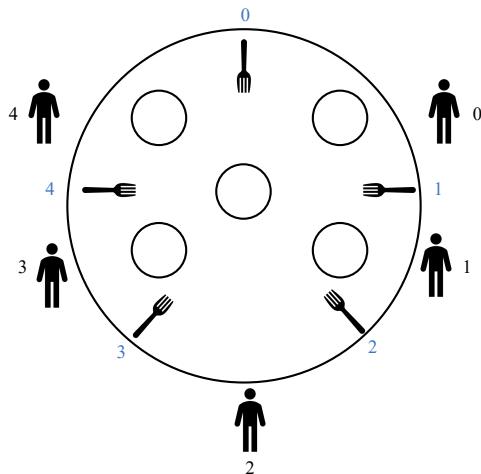
```

1 int A[m];
2 semaphore s[m];
3 for (int i = 0; i < m; i++){
4     s[i] = 1;
5 }
6 cobegin
7     process Pi(){
8         int Xi;
9         L1: 按旅客订票要求找到A[j];
10        P(s[j]);
11        Xi = A[j];
12        if (Xi >= 1){
13            Xi = Xi - 1;
14            A[j] = Xi;
15            V(s[j]);
16            {输出一张票};
17        }else{
18            V(s[j]);
19            {输出票已售完};
20        }
21        goto L1;
22    }
23 coend

```

### 3.1.2.2 哲学家就餐问题

问题描述：有五个哲学家围坐在一圆桌旁，桌中央有一盘通心面，每人面前有一只空盘子，每两人之间放一把叉子。每个哲学家思考、饥饿、然后吃通心面。为了吃面，每个哲学家必须获得两把叉子，且每人只能直接从自己左边或右边去取叉子



```

1 //存在死锁的哲学家就餐问题
2 semaphore fork[5];
3 for (int i = 0; i < 5; i++){
4     fork[i] = 1;
5 }
6 cobegin
7 process philosopher_i( ) { //i=0,1,2,3,4
8     while(true) {
9         think();
10        P(fork[i]);           //先取右手的叉子
11        P(fork[(i+1)%5]);   //再取左手的叉子
12        eat();
13        V(fork[i]);
14        V(fork[(i+1)%5]);
15    }
16 }
17 coend

```

上面的代码存在死锁的问题：比如当每一个哲学家都拿到了一侧的叉子将出现死锁，有若干种办法可避免死锁

- 至多允许四个哲学家同时取叉子（C.A.R.Hoare 方案）
- 奇数号的哲学家先取左手边的叉子，偶数号的哲学家先取右手边的叉子

“至多允许四个哲学家同时取叉子”的解决办法

```

1 //设置侍者，添加房间信号量来控制同时取叉子的哲学家个数
2 semaphore fork[5];
3 for (int i = 0; i < 5; i++){
4     fork[i] = 1;
5 }
6 semaphore room = 4; //增加一个侍者，设想有两个房间，1号房间是会议室，2号房间是餐厅
7 cobegin
8 process philosopher_i(){ /*i=0,1,2,3,4 */
9     while(true) {
10         think();

```

```

11     P(room); //控制最多允许4位哲学家进入2号房间餐厅取叉子
12     P(fork[i]);
13     P(fork[(i+1)%5]) ;
14     eat();
15     V(fork[i]);
16     V(fork[(i+1)%5]);
17     V(room);
18 }
19 }
20 coend

```

“奇数号的哲学家先取左手边的叉子，偶数号的哲学家先取右手边的叉子”的解决办法

```

//限制奇数号的哲学家优先取左手边的叉子，偶数号的哲学家优先取右手边的叉子
void philosopher(int i){
    if (i % 2 == 0){
        P(fork[i]);           //偶数哲学家先取右手边的叉子
        P(fork[(i+1)%5]);   //后取左手边的叉子
        eat();
        V(fork[i]);
        V(fork[(i+1)%5]);
    }else{
        P(fork[(i+1)%5]); //奇数号哲学家先取左手边的叉子
        P(fork[i]);         //后取右手边的叉子
        eat();
        V(fork[(i+1)%5]);
        V(fork[i]);
    }
}

```

## 3.2 PV 操作与进程同步

### 3.2.1 PV 操作解决进程同步问题的基本思路

使用信号量与 PV 操作解决进程同步问题的基本思路如下：

- 定义一个记录型信号量，用于表示可用的消息数（资源数）
- 等待消息的进程在执行 P 操作后，如果没有得到消息，它就会被阻塞在同步信号量的等待队列，并依序排队
- 发出消息的进程在执行 V 操作后，如果有等待该消息的进程，它将唤醒一个在该同步信号量上等待的进程

### 3.2.2 生产者消费者问题

问题描述：有  $n$  个生产者和  $m$  个消费者，连接在一个有  $k$  个单位缓冲区的有界缓冲上。其中，生产者进程 `Producer_i` 和消费者进程 `Consumer_j` 都是并发进程，只要缓冲区未满，生产者 `Producer_i` 生产的产品就可投入缓冲区；只要缓冲区不空，消费者进程 `Consumer_j` 就可从缓冲区取走并消耗产品。

可能的情形有：

- $n = 1, m = 1, k = 1$

- $n = 1, m = 1, k > 1$
- $n > 1, m > 1, k > 1$

### 3.2.2.1 生产者消费者问题分析

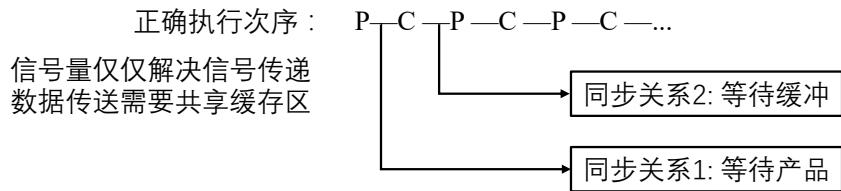
```

int B; //缓存区，用于存放产品

process producer(){ //生产者进程
    L1: produce a product; //生产产品
    B = product; //放产品
    goto L1;
}

process consumer(){ //消费者进程
    L2: product = B; //取产品
    consume a product; //消费产品
    goto L2;
}

```



#### 生产者和消费者共享缓冲区

- 缓冲区有空位时，生产者可放入产品，否则等待缓冲区有产品时，消费者可取出产品，否则等待
- 同步问题的解决思路
- 同步关系 1：对于消费者，设置一个同步信号量 sget 等待产品。在初始状态时，缓冲区为空，因此该同步信号量的初始值为 0。V(sget) 操作位于生产者进程中，表示产品已放置在缓冲区中；P(sget) 位于消费者进程中，表示需要等待生产者填入产品
  - 同步关系 2：设置另一信号量 sput，表示等待缓冲区为空。在初始状态时，缓冲区为空，因此该同步信号量的初始值为 1，即生产者初始时可以放进一件产品。P(sput) 操作位于生产者进程中，表示需要等待缓冲区为空时才能生产产品；V(sput) 位于消费者进程，表示消费者已取出缓冲区中的产品。

### 3.2.2.2 一个生产者、一个消费者共享一个缓冲区问题

```

int B; //共享缓冲区
semaphore sput; //可以使用的空缓冲区数
semaphore sget; //缓冲区内可以使用的产品数
sput = 1; //缓冲区内允许放入一件产品
sget = 0; //缓冲区内没有产品
cobegin
    process producer(){
        L1: produce a product;
        P(sput);
        B = product;
        V(sget);
        goto L1;
    }
    process consumer(){
        L2: P(sget);
        product = B;
        V(sput);
        consume a product;
        goto L2;
    }
coend

```

### 3.2.2.3 一个生产者、一个消费者共享多个缓冲区问题

```

int B[k];           //共享缓冲区队列
semaphore sput;    //可以使用的空缓冲区数
semaphore sget;    //缓冲区内可以使用的产品数
sput = k;          //缓冲区内允许放入k件产品
sget = 0;          //缓冲区内没有产品
int putptr, getptr; //循环队列指针，分别用于表示放产品和取产品的位置
putptr = 0;
getptr = 0;
cobegin
    process producer_i(){
        L1: produce a product;
        P(sput);
        B[putptr] = product;
        putptr = (putptr + 1) % k
        V(sget);
        goto L1;
    }
    process consumer_j(){
        L2: P(sget);
        product = B[getptr];
        getptr = (getptr + 1) % k;
        V(sput);
        consume a product;
        goto L2;
    }
coend

```

### 3.2.2.4 多个生产者、多个消费者共享多个缓冲区问题

```

int B[k];           //共享缓冲区队列
semaphore sput;    //可以使用的空缓冲区数
semaphore sget;    //缓冲区内可以使用的产品数
sput = k;          //缓冲区内允许放入k件产品
sget = 0;          //缓冲区内没有产品
int putptr, getptr; //循环队列指针，分别用于表示放产品和取产品的位置
putptr = 0; getptr = 0;
semaphore s1, s2;  //消费者互斥使用putptr，生产者互斥使用getptr
s1 = 1; s2 = 1;
cobegin
    process producer_i(){
        L1: produce a product;
        P(sput);
        P(s1);
        B[putptr] = product;
        putptr = (putptr + 1) % k
        V(s1);
        V(sget);
        goto L1;
    }
    process consumer_j(){
        L2: P(sget);
        P(s2);
        product = B[getptr];
        getptr = (getptr + 1) % k;
        V(s2);
        V(sput);
        consume a product;
        goto L2;
    }
coend

```

### 3.2.3 苹果-橘子问题

问题描述：桌上有一只盘子，每次只能放入一个水果。爸爸专门向盘子中放苹果，妈妈专门向盘子中放橘子，一个儿子专门等着吃盘子中的橘子，一个女儿专门等着吃盘子中的苹果。

```

int plate;
semaphore sp; // 盘子里可以放几个水果
semaphore sg1; // 盘子里有橘子
semaphore sg2; // 盘子里有苹果
sp = 1; // 盘子里允许放入一个水果
sg1 = 0; // 盘子里没有橘子
sg2 = 0; // 盘子里没有苹果
cobegin
    process father(){
        L1: 削一个苹果
        P(sp);
        将苹果放入plate;
        V(sg2);
        goto L1;
    }
    process son(){
        L3: P(sg1);
        从plate中取橘子;
        V(sp);
        吃橘子;
        goto L3;
    }
    process mother(){
        L2: 剥一个橘子
        P(sp);
        将橘子放入plate;
        V(sg1);
        goto L2;
    }
    process daughter(){
        L4: P(sg2);
        从plate中取苹果;
        V(sp);
        吃苹果;
        goto L4;
    }
coend

```

## 4 信号量与 PV 操作习题

### 4.1 读者/写者问题

读者与写者问题 (Courtois, 1971): 有两组并发进程: 读者和写者, 共享一个文件  $F$ , 要求:

1. 允许多个读者可同时对文件执行读操作
2. 只允许一个写者往文件中写信息
3. 任意写者在完成写操作之前不允许其他读者或写者工作
4. 写者执行写操作前, 应让已有的写者和读者全部退出

#### 4.1.1 读者优先

```

1 semaphore rmutex, wmutex;
2 rmutex = 1; wmutex = 1;
3 int readcount = 0; // 读进程计数
4
5 process reader_i(){
6     while(true){
7         P(rmutex); // 上锁进行读者计数更新, 防止多个读者计数发生冲突的问题
8         readcount++;
9         if(readcount == 1) P(wmutex); // 如果自己是第一个读者, 则对文件上锁
10        V(rmutex); // 开锁允许其他读者
11        读文件;
12        P(rmutex); // 上锁进行读者计数更新, 防止多个读者计数发生冲突的问题
}

```

```

13         readcount--; //因为已经完成读操作，所以读者计数减1
14         if(readcount == 0) V(wmutex); //如果自己是最后一个读者，则释放文件资源
15         V(rmutex);
16     }
17 }
18
19 process writer_i(){
20     while(true){
21         P(wmutex);           //文件上锁
22         写文件;
23         V(wmutex);          //开锁允许其他进程的读或写操作
24     }
25 }
```

该程序的设计思路是第一个读者对文件执行占用，最后一个读者对文件执行释放。也正是由于最后一个读者才会释放文件资源，所以当有写者的时候，写者会被阻塞直到所有的读者完成操作。因此这种解法的问题是可能会饿死写者，该解法是读者优先的。

#### 4.1.2 写者优先

```

1 int readcount = 0, writecount = 0;
2 semaphore x = 1, y = 1, z = 1;
3 semaphore rmutex = 1, wmutex = 1;
4
5 process reader_i(){
6     while(true){
7         P(z);
8         P(rmutex);
9         P(x);
10        readcount++;
11        if(readcount == 1) P(wmutex);
12        V(x);
13        V(rmutex);
14        V(z);
15        读文件;
16        P(x);
17        readcount--;
18        if (readcount == 0)      V(wmutex);
19        V(x);
20    }
21 }
22
23 process writer_i(){
24     while(true){
25         P(y);
26         writecount++;
27         if(writecount == 1) P(rmutex);
28         V(y);
29         P(wmutex);
30         写操作;
```

```

31     V(wmutex);
32     P(y);
33     writecount--;
34     if(writecount == 0) V(rmutex);
35     V(y);
36 }
37 }
```

#### 4.1.3 读写平衡

```

1 semaphore rmutex, wmutex, S;
2 rmutex = 1; wmutex = 1; S = 1;
3 int readcount = 0;
4 process reader_i(){
5     while(true){
6         P(S);
7         P(rmutex);
8         readcount++;
9         if(readcount == 1) P(wmutex);
10        V(rmutex);
11        V(S);
12        读文件;
13        P(rmutex);
14        readcount--;
15        if(readcount == 0)      V(wmutex);
16        V(rmutex);
17    }
18 }

19
20 process writer_i(){
21     while(true){
22         P(S);
23         P(wmutex);
24         写文件;
25         V(wmutex);
26         V(S);
27     }
28 }
```

S 一次对外面提供一个服务的空位，不管是读者还是写者都需要先抢占这个空位

## 4.2 睡眠的理发师问题

问题描述：理发店理有一位理发师、一把理发椅和  $n$  把供等候理发的顾客坐的椅子

- 如果没有顾客，理发师便在理发椅上睡觉
- 一个顾客到来时，它必须叫醒理发师
- 如果理发师正在理发时又有顾客来到，则如果有空椅子可坐，就坐下来等待，否则就离开

```

int waiting = 0; //等待理发的顾客坐的椅子数
int chairs = n; //为顾客准备的椅子数
semaphore customers, barbers, mutex;
customers = 0; barbers = 0; mutex = 1;

process barber(){
    while(true){
        P(customers); //判断有无顾客，若无顾客，则理发师睡眠
        P(mutex); //若有顾客，则进入临界区
        waiting--;
        V(barbers); //理发师准备为顾客理发
        V(mutex); //退出临界区
        为顾客理发;
    }
}

process customer_i(){
    P(mutex); //进入临界区
    if(waiting < chairs){ //有无空椅子
        waiting++;
        V(customers); //唤醒理发师
        V(mutex); //退出临界区
        P(barbers); //理发师忙，顾客坐下等待
        理发;
    }else{
        V(mutex); //等待人满，顾客离开
    }
}

```

### 4.3 农夫猎人问题

问题描述：有一个铁笼子，每次只能放入一个动物。猎手向笼中放入老虎，农夫向笼中放入羊，动物园等待取笼中的老虎，饭店等待取笼中的羊。

```

semaphore Scage = 1;
semaphore Stiger = 0;
semaphore Ssheep = 0;

void hunter(){
    while(true){
        ...
        P(Scage);
        将虎放入笼中;
        V(Stiger);
    }
}

void peasant(){
    while(true){
        ...
        P(Scage);
        将羊放入笼中;
        V(Ssheep);
    }
}

void hotel(){
    while(true){
        P(Ssheep);
        将羊取出笼中;
        V(Scage);
        ...
    }
}

void zoo(){
    while(true){
        P(Stiger);
        将虎取出笼中;
        V(Scage);
        ...
    }
}

void main(){
    parbegin(hunter, peasant, hotel, zoo);
}

```

### 4.4 银行业务问题

问题描述：某大型银行办理人民币储蓄业务，由  $n$  个储蓄员负责。每个顾客进入银行后先至取号机取一个号，并且在等待区找到空沙发坐下等着叫号。取号机给出的号码依次递增，并假定有足够的空沙发容纳顾客。当一个储蓄员空闲下来，就叫下一个号。

```

semaphore customer_count, server_count, mutex;
customer_count = 0; server_count = 0; mutex = 1;

process customer_i(){
    取号;
    P(mutex);
    等待区找到空沙发坐下;
    V(mutex);
    V(customer_count);
    P(server_count);
}

process server_j(){
    L: P(customer_count);
    P(mutex);
    被叫到号的顾客离开沙发走出等待区;
    V(mutex);
    为该号客人服务;
    客人离开;
    V(server_count);
    goto L;
}

```

## 4.5 缓冲区管理

问题描述：有  $n$  个进程将字符逐个读入到一个容量为 80 的缓冲区中 ( $n > 1$ )，当缓冲区满后，由输出进程  $Q$  负责一次性取走这 80 个字符，这种过程循环往复。

```

semaphore mutex, empty, full;
int count, in;
char[80] buffer;
mutex = 1; empty = 80; full = 0;
count = 0; in = 0;

process Pi(){
    L: 读入一个字符到x;
    P(empty);
    P(mutex);
    buffer[in] = x;
    in = (in + 1) % 80;
    count++;
    if(count == 80){
        count = 0;
        V(mutex);
        V(full);
    }else{
        V(mutex);
    }
}

process Q(){
    while(true){
        P(full);
        P(mutex);
        for(int i = 0; i < 80; i++)
            read buffer[i];
        in = 0;
        V(mutex);
        for(int i = 0; i < 80; i++){
            V(empty);
        }
    }
}

```

## 4.6 售票问题

问题描述：汽车司机与售票员之间必须协同工作，一方面只有售票员把车门关好了司机才能开车，因此，售票员关好门应通知司机开车，然后售票员进行售票。另一方面，只有当汽车已经停下，售票员才能开门上下客，故司机停车后应该通知售票员。

现假定某辆公共汽车上有一名司机与两名售票员，汽车当前正在始发站停车上客。

```

semaphore run1, run2, stop1, stop2;
run1 = 0; run2 = 0; stop1 = 0; stop2 = 0;

void Driver(){
    while(true){
        P(run1);
        P(run2);
        开车;
        停车;
        V(stop1);
        V(stop2);
    }
}

void Seller1(){
    while(true){
        上乘客;
        关车门;
        V(run1);
        售车票;
        P(stop1);
        开车门;
        下乘客;
    }
}

void Seller2(){
    while(true){
        上乘客;
        关车门;
        V(run2);
        售车票;
        P(stop2);
        开车门;
        下乘客;
    }
}

void main(){
    parbegin(Driver, Seller1, Seller2);
}

```

## 4.7 吸烟者问题

问题描述：三个吸烟者在一个房间内，还有一个香烟供应者。为了制造并抽掉香烟，每个吸烟者需要三样东西：烟草、纸和火柴，供应者有丰富货物提供。三个吸烟者中，第一个有自己的烟草，第二个有自己的纸和第三个有自己的火柴。供应者随机地将两样东西放在桌子上，允许一个吸烟者进行对健康不利的吸烟。当吸烟者完成吸烟后唤醒供应者，供应者再把两样东西放在桌子上，唤醒另一个吸烟者。

```

semaphore sput, sget[3];
sput = 1; sget[i] = 0; //i=0,1,2

process businessman(){
    L1: i = RAND() % 3;
    j = RAND() % 3;
    if(i == j)
        goto L1;
    P(sput);
    put items[i] on the table;
    put items[j] on the table;
    if((i == 0 && j == 1) || (i == 1 && j == 0))
        V(sget[2]);
    if((i == 1 && j == 2) || (i == 2 && j == 1))
        V(sget[0]);
    if((i == 0 && j == 2) || (i == 2 && j == 0))
        V(sget[1]);
    goto L1;
}

process consumer(k){
    L2: P(sget[k]);
    take one item from the table;
    take one item from the table;
    V(sput);
    make cigarette and smoke;
    goto L2;
}

```

## 4.8 独木桥问题

### 4.8.1 独木桥问题 1

问题描述：东西向汽车过独木桥，为了保证安全，只要桥上无车，则允许一方的汽车过桥，待一方的车全部过完后，另一方的车才允许过桥。

```

semaphore wait, mutex1, mutex2;
mutex1 = 1; mutex2 = 1; wait = 1;
int count1, count2; count1 = 0; count2 = 0;

process P东(){
    {
        P(mutex1);
        count1++;
        if(count1 == 1)
            P(wait);
        V(mutex1);
        过独木桥;
        P(mutex1);
        count1--;
        if(count1 == 0)
            V(wait);
        V(mutex1);
    }
}

process P西(){
    {
        P(mutex2);
        count2++;
        if(count2 == 1)
            P(wait);
        V(mutex2);
        过独木桥;
        P(mutex2);
        count2--;
        if(count2 == 0)
            V(wait);
        V(mutex2);
    }
}

```

### 4.8.2 独木桥问题 2

问题描述：在独木桥问题 1 的基础上，限制桥面上最多可以有  $k$  辆汽车通过。

```

semaphore wait, mutex1, mutex2, bridge;
mutex1 = 1; mutex2 = 1; wait = 1; bridge = k;
int count1, count2; count1 = 0; count2 = 0;

process P东(){
    P(mutex1);
    count1++;
    if(count1 == 1)
        P(wait);
    V(mutex1);
    P(bridge);
    过独木桥;
    V(bridge);
    P(mutex1);
    count1--;
    if(count1 == 0)
        V(wait);
    V(mutex1);
}

process P西(){
    P(mutex2);
    count2++;
    if(count2 == 1)
        P(wait);
    V(mutex2);
    P(bridge);
    过独木桥;
    V(bridge);
    P(mutex2);
    count2--;
    if(count2 == 0)
        V(wait);
    V(mutex2);
}

```

### 4.8.3 独木桥问题 3

问题描述：在独木桥问题 1 的基础上，以 3 辆汽车为一组，要求保证东方和西方以组为单位交替通过汽车。

```

semaphore wait, mutex1, mutex2;
mutex1 = 1; mutex2 = 1; wait = 1;
int countu1, countu2, countd1, countd2; countu1 = 0; countu2 = 0; countd1 = 0; countd2 = 0;

process P东(){
    P(S1);
    P(mutex1);
    countu1++;
    if(countu1 == 1)
        P(wait);
    V(mutex1);
    过独木桥;
    V(S2);
    P(mutex1);
    countu1--; countd1++;
    if((countu1 == 0) && (countd1 == 3))
        countd1 = 0; V(wait);
    V(mutex1);
}

process P西(){
    P(S2);
    P(mutex2);
    countu2++;
    if(countu2 == 1)
        P(wait);
    V(mutex2);
    过独木桥;
    V(S1);
    P(mutex2);
    countu2--; countd2++;
    if((countu2 == 0) && (countd2 == 3))
        countd2 = 0; V(wait);
    V(mutex2);
}

```

### 4.8.4 独木桥问题 4

问题描述：在独木桥问题 1 的基础上，要求各方向的汽车串行过桥，但当另一方提出过桥时，应能阻止对方未上桥的后继车辆，待桥面上的汽车过完桥后，另一方的汽车开始过桥。

```

semaphore stop, wait, mutex1, mutex2;
mutex1 = 1; mutex2 = 1; stop = 1; wait = 1;
int count1, count2; count1 = 0; count2 = 0;

process P东(){
    P(stop);
    P(mutex1);
    count1++;
    if(count1 == 1)
        P(wait);
    V(mutex1);
    V(stop);
    过独木桥;
    P(mutex1);
    count1--;
    if(count1 == 0)
        V(wait);
    V(mutex1);
}

process P西(){
    P(stop);
    P(mutex2);
    count2++;
    if(count2 == 1)
        P(wait);
    V(mutex2);
    V(stop);
    过独木桥;
    P(mutex2);
    count2--;
    if(count2 == 0)
        V(wait);
    V(mutex2);
}

```

## 5 管程

### 5.1 管程概述

#### 5.1.1 管程的提出

管程是由若干公共变量及其说明和所有访问这些变量的过程所组成

- 管程把分散在各个进程中互斥地访问公共变量的那些临界区集中起来管理
- 管程的局部变量只能由该管程的过程存取

管程的属性：

1. 共享性：管程中的移出过程可被所有调用管程的过程的进程所共享
2. 安全性：管程的局部变量只能由此管程内部分访问，不允许进程或其他管程直接访问
3. 互斥性：任一时刻，共享资源的进程可以访问管程中的管理此资源的过程，但最多只有一个调用者能够真正进入管程，其他调用者必须等待直到管程可用

#### 5.1.2 管程的规格定义与实现思路

每个管程都有一个名字以供标识，其一般形式为：

```

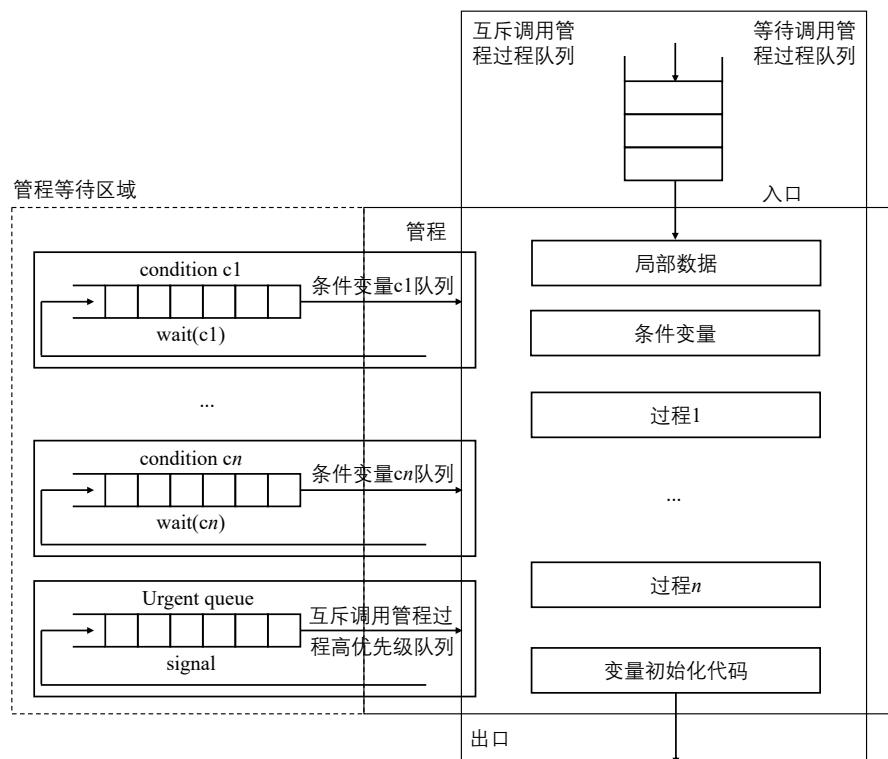
1 type <管程名> = monitor {
2     局部变量说明;
3     条件变量说明;
4     初始化语句;
5     define <(能被其他模块引用的)过程名列表>;
6     use <(要引用的模块外定义的)过程名列表>;
7     procedure <过程名>(<形式参数表>);
8
9     过程名/函数名(<形式参数表>) {
10         <过程/函数体>;

```

```

11 }
12 ...
13
14 过程名/函数名(形式参数表) {
15   <过程/函数体>;
16 }
17 }
18 }
```

管程的执行模型：



条件变量是出现在管程内的一种数据结构，且只有在管程中才能被访问，它对管程内的所有过程是全局的，只能通过两个原语操作来控制它：

- **wait():** 阻塞调用进程并释放管程，直到另一个进程在该条件变量上执行 signal()
- **signal():** 如果存在其他进程由于对条件变量执行 wait() 而被阻塞，便释放之；如果没有进程在等待，那么则信号不被保存

使用 signal 释放等待进程时，可能出现两个进程同时停留在管程内，解决方法有：

- 执行 signal 的进程等待，直到被释放进程退出管程或等待另一个条件变量
  - 霍尔 (Hoare) 采用该办法
- 被释放进程等待，直到执行 signal 的进程退出管程或等待另一个条件
  - 汉森 (Hansen) 选择两者的折衷，规定管程中的过程所执行的 signal 操作是过程体的最后一个操作

管程和进程的对比：

- 管程定义公用数据结构，进程定义私有数据结构
- 管程将共享变量上的同步操作集中统一管理，临界区分散在每个进程中
- 管程为了解决进程共享资源的互斥，进程为了占有系统资源和实现系统并发
- 管程被想要使用共享资源的所有进程调用，管程和调用它的进程不能并行工作；进程间可以并行工作

## 5.2 霍尔管程

霍尔管程是一种更具普适性的管程实现方法，它使用 PV 操作原语实现对管程中过程的互斥调用功能，同时实现对共享资源互斥使用的管理。

- 每当有进程等待资源时，霍尔管程将让执行 signal 操作的进程阻塞自己，直到它被释放的进程退出管程或产生了其他的等待条件为止
- 与汉森管程的实现方法相比，该方法不要求 signal 操作是过程体的最后一个操作，且 wait 和 signal 操作可被设计成两个可以中断的过程，而非原语
- 霍尔管程可以通过操作系统的程序库或高级程序设计语言，在基础操作系统的 PV 操作原语上实现，而不需要扩展操作系统的内核

### 5.2.1 霍尔管程的数据结构

```

1  typedef struct InterfaceModule{ //InterfaceModule是结构体名字
2      semaphore mutex;    //进程调用管程过程前使用的互斥信号量
3      semaphore next;     //发出signal的进程阻塞自己的信号量
4      int next_count;    //在next上等待的进程数
5  };
6  mutex=1; next=0; next_count=0; //初始化语句

```

- **mutex**
  - 对每个管程，使用用于管程中过程互斥调用的信号量mutex（初值为1）
  - 进程调用管程中的任何过程时，应执行P(mutex)；进程退出管程时，需要判断是否有进程在next信号量等待，如果有（即next-count>0），则通过V(next)唤醒一个发出signal的进程，否则应执行V(mutex)开放管程，以便让其他调用者进入
  - 为了使进程在等待资源期间，其他进程能进入管程，故在wait操作中也必须执行V(mutex)，否则会妨碍其他进程进入管程，导致无法释放资源
- **next和next-count**
  - 对每个管程，引入信号量next（初值为0），凡发出signal操作的进程应该用P(next)阻塞自己，直到被释放进程退出管程或产生其他等待条件
  - 进程在退出管程的过程前，须检查是否有别的进程在信号量next上等待，若有，则用V(next)唤醒它。
  - next-count（初值为0），用来记录在next上等待的进程个数
- **x-sem和x-count**
  - 引入信号量x-sem（初值为0），申请资源得不到满足时，执行P(x-sem)阻塞。由于释放资源时，需要知道是否有别的进程在等待资源，用计数器x-count（初值为0）记录等待资源的进程数
  - 执行signal操作时，应让等待资源的诸进程中的某个进程立即恢复运行，而不让其他进程抢先进入管程，这可以用V(x-sem)来实现

### 5.2.2 霍尔管程的 enter() 和 leave() 操作

```

1 void enter(InterfaceModule &IM) {
2     P(IM.mutex);           //判断是否有发出过signal的进程
3 }
4
5 void leave(InterfaceModule &IM) {
6     if(IM.next_count > 0)
7         V(IM.next);        //有就释放一个发出过signal的进程
8     else
9         V(IM.mutex);       //否则开放管程
10}

```

### 5.2.3 霍尔管程的 wait() 操作

```

1 void wait(semaphore &x_sem, int &x_count, InterfaceModule &IM) {
2     x_count++;            //等资源进程个数加1, x_count初始化为0
3     if (IM.next_count > 0) //判断是否有发出过signal的进程
4         V(IM.next);       //有就释放一个
5     else
6         V(IM.mutex);      //否则开放管程
7     P(x_sem);             //等资源进程阻塞自己, x_sem初始化为0
8     x_count--;            //等资源进程个数减1
9 }

```

### 5.2.4 霍尔管程的 signal() 操作

```

1 void signal(semaphore &x_sem, int &x_count, InterfaceModule &IM) {
2     if(x_count > 0){      //判断是否有等待资源的进程
3         IM.next_count++; //发出signal进程个数加1
4         V(x_sem);        //释放一个等资源的进程
5         P(IM.next);       //发出signal进程阻塞自己
6         IM.next_count--; //发出signal进程个数减1
7     }
8 }

```

## 5.3 霍尔管程求解进程互斥与同步问题

### 5.3.1 霍尔管程求解读者/写者问题

```

1 type read-write = monitor
2     int rc, wc;
3     semaphore R, W; R = 0; W = 0;
4     int R_count, W_count;
5     rc = 0; wc = 0;
6     InterfaceModule IM;
7     DEFINE start_read, end_read, start_write, end_write;
8     USE wait, signal, enter, leave;

```

<pre> void start_read( ){     enter(IM);     if(wc &gt; 0)         wait(R, R_count, IM);     rc++;     signal(R, R_count, IM);     leave(IM); } </pre>	<pre> void start_write( ){     enter(IM);     wc++;     if(rc &gt; 0    wc &gt; 1)         wait(W, W_count, IM);     leave(IM); } </pre>
<pre> void end_read( ){     enter(IM);     rc--;     if(rc == 0)         signal(W, W_count, IM);     leave(IM); } </pre>	<pre> void end_write( ){     enter(IM);     wc--;     if(wc &gt; 0)         signal(W, W_count, IM);     else         signal(R, R_count, IM);     leave(IM); } </pre>
<pre> process P1( ){ ...     read-write.start_read( );     {read};     read-write.end_read( ); ... } </pre>	<pre> process P2( ){ ...     read-write.start_write( );     {write};     read-write.end_write( ); ... } </pre>

### 5.3.2 霍尔管程求解哲学家就餐问题

```

1 type dining_philosophers = monitor
2     enum {thinking, hungry, eating} state[5];
3     semaphore self[5]; int self_count[5];
4     InterfaceModule IM; for (int i=0; i<5; i++){state[i] = thinking; //初始化, i为进程号}
5     define pickup, putdown;
6     use enter, leave, wait, signal;

```

<pre> void pickup(int i) { //i=0,1,...,4     enter(IM);     state[i] = hungry;     test(i);     if(state[i] != eating)         wait(self[i], self_count[i], IM);     leave(IM); } </pre>	<pre> void putdown(int i) { //i=0,1,2,...4     enter(IM);     state[i] = thinking;     test((i - 1) % 5);     test((i + 1) % 5);     leave(IM); } </pre>
<pre> void test(int k) { //k=0,1,...,4     if((state[(k-1)%5] != eating) &amp;&amp; (state[k] == hungry) &amp;&amp; (state[(k+1)%5] != eating)) {         state[k] = eating;         signal(self[k], self_count[k], IM);     } } </pre>	

### 5.3.3 霍尔管程求解生产者消费者问题

```

1 type producer_consumer = monitor
2     item B[k]; int in, out; //B[k]表示缓冲单元, in和out是存取指针

```

```

3 int count; //缓冲中产品数
4 semaphore notfull, notempty; //条件变量
5 int notfull_count, notempty_count;
6 InterfaceModule IM;
7 define append, take;
8 use enter, leave, wait, signal;

```

<pre> void append(item x){     enter(IM);     if(count == k) //缓冲已满         wait(notfull, notfull_count, IM);     B[in] = x;     in = (in + 1) % k;     count++; //增加一个产品     //唤醒等待消费者     signal(notempty, notempty_count, IM);     leave(IM); } </pre>	<pre> void take(item &amp;x){     enter(IM);     if(count == 0) //缓冲已空         wait(notempty, notempty_count, IM);     x = B[out];     out = (out + 1) % k;     count--; //减少一个产品     //唤醒等待生产者     signal(notfull, notfull_count, IM);     leave(IM); } </pre>
<pre> process producer_i(){//i=1,...,n     item x;     produce(x);     producer_consumer.append(x) } </pre>	<pre> process consumer_j(){//j=1,...m     item x;     producer_consumer.take(x);     consume(x); } </pre>

### 5.3.4 霍尔管程求解苹果桔子问题

```

1 type FMSD = monitor
2     enum FRUIT {apple, orange} plate; bool full;
3     semaphore SP, SS, SD; int SP_count, SS_count, SD_count; full = false
4     InterfaceModule IM;
5     define put, get;
6     use enter, leave, wait, signal;

```

<pre> void put(FRUIT fruit){     enter(IM);     if(full)         wait(SP, SP_count, IM);     full = true; plate = fruit;     if(fruit == orange)         signal(SS, SS_count, IM);     else         signal(SD, SD_count, IM);     leave(IM); } </pre>	<pre> void get(FRUIT fruit, FRUIT &amp;x){     enter(IM);     if (!full    plate != fruit){         if (fruit == orange)             wait(SS, SS_count, IM);         else             wait(SD, SD_count, IM);     }     x = plate;     full = false;     signal(SP, SP_count, IM);     leave(IM); } </pre>
---	--

<pre>process father(){     {准备好苹果};     FMSD.put(apple); }</pre>	<pre>process son( ){     FMSD.get(orange, x);     {吃取到的桔子}; }</pre>
<pre>process mother( ){     {准备好桔子};     FMSD.put(orange); }</pre>	<pre>process daughter( ){     FMSD.get(apple, x);     {吃取到的苹果}; }</pre>

## 6 进程通信

### 6.1 进程通信概述

交往进程通过信号量操作实现进程互斥和同步，这是一种低级通信方式

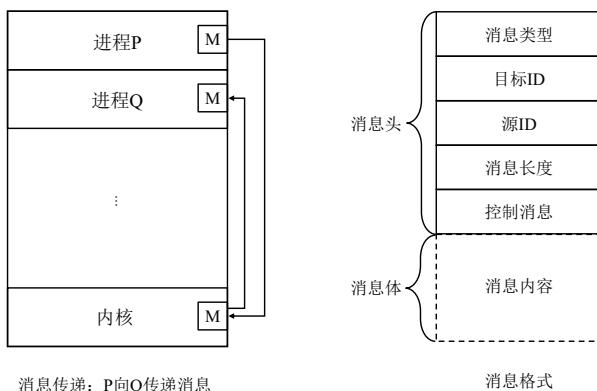
进程有时还需要交换更多的信息（如把数据传送给另一个进程），为此可以引进高级通信方式：进程通信机制，实现进程间用信件来交换信息，进程通信扩充了并发进程的数据共享

- 通过该通信机制，一个正在执行的进程可以在任何时刻向另一个正在执行的进程发送或请求信件
- 如果一个进程在某一时刻的执行依赖于另一进程的信件或等待其他进程对发送信件的回答，那么通信机制将与进程阻塞和释放紧密联系
- 这样，进程间的通信就进一步扩充了并发进程间对数据的共享

#### 6.1.1 进程直接通信

在进程直接通信方式下，企图发送或接收消息的每个进程必须指出信件发给谁或从谁那里接收信件，可用下面两条原语来实现进程之间的通信

- `send(P, 信件)`：将信件发送给进程 P
- `receive(Q, 信件)`：从进程 Q 接收信件



#### 6.1.2 进程间接通信

对于间接通信方式，进程间发送或接收信件通过一个共享的数据结构——信箱来进行，每个信箱有一个唯一的标识符

- 信箱是存放信件的存储区域，可以分成信箱头和信箱体两部分
  - 信箱头指出信箱容量、信件格式、存放信件位置的指针等
  - 信箱体用来存放信件，信箱体分成若干个区，每个区可容纳一封信
- 当两个以上的进程共享一个信箱时，它们之间通过共享的信箱进行通信

间接通信方式中的发送和接收原语形式如下：

- `send(A, 信件)`: 将信件发送到信箱 A
- `receive(A, 信件)`: 从信箱 A 接收信件

发送和接收两条原语的功能如下：

- 发送信件：如果指定的信箱未满，则将信件送入信箱中由指针所指示的位置，并释放等待该信箱中信件的等待者；否则，发送信件者被置成等待信箱状态
- 接收信件：如果指定信箱中有信，则取出一封信件，并释放等待信箱的等待者；否则，接收信件者被置成等待信箱中信件的状态

`send/receive`原语的算法描述：

<pre> type struct box{     int size; //信箱大小     int count; //现有信件数     message letter[]; //信箱中的信件     semaphore S1, S2; //等信箱和等信件信号量 }  procedure send(box B, message M){     int i;     if(B.count == B.size)         W(B.S1);     i = B.count + 1;     B.letter[i] = M;     B.count = i;     R(B.S2); } </pre>	<pre> procedure receive(box B, message X){     int i;     if(B.count == 0)         W(B.S2);     B.count = B.count - 1;     X = B.letter[1];     if(B.count != 0){         for(int i = 0; i &lt; B.count; i++){             B.letter[i] = B.letter[i+1];         }     }     R(B.S1); } </pre>
--	---

利用消息传递求解生产者消费者问题

```

1 creat_mailbox(producer); //创建信箱
2 creat_mailbox(consumer);

3

4 void producer_i(){ //i=1,...,n
    message pmsg;
    while(true){
        pmsg = produce();
        send(consumer, pmsg);
    }
}

11 void consumer_j(){ //j=1,...,m

```

```

13     message cmsg;
14     while(true){
15         receive (consumer, cmsg);
16         consume(cmsg);
17     }
18 }
19
20 cobegin
21     producer_i();
22     consumer_j();
23 coend

```

### 6.1.3 消息缓冲通信

消息缓冲是在 1973 年由 P.B.Hansan 提出的一种进程间高级通信设施，并在 RC4000 系统中实现。消息缓冲通信的基本思想：

- 由操作系统统一管理一组用于通信的消息缓冲存储区，每一个消息缓冲存储区可存放一个消息（信件）
- 当一个进程要发送消息时，先在自己的消息发送区里生成待发送的消息，包括：接收进程名、消息长度、消息正文等
- 然后，向系统申请一个消息缓冲区，把消息从发送区复制到消息缓冲区中，注意在复制过程中系统会将接收进程名换成发送进程名，以便接收者识别
- 随后该消息缓冲区被挂到接收消息的进程的消息队列上，供接收者在需要时从消息队列中摘下并复制到消息接收区去使用，同时释放消息缓冲区

消息缓冲通信涉及的数据结构：

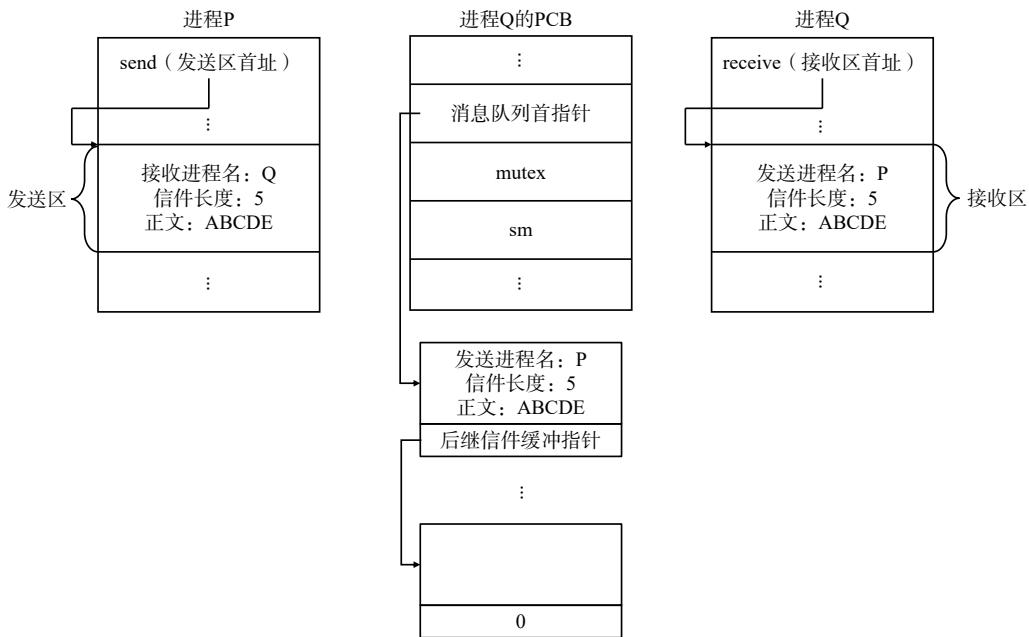
- **sender**: 发送消息的进程名或标识符
- **size**: 发送的消息长度
- **text**: 发送的消息正文
- **next\_ptr**: 指向下一个消息缓冲区的指针

在进程的 PCB 中涉及通信的数据结构：

- **mptr**: 消息队列队首指针
- **mutex**: 消息队列互斥信号量，初值为 1
- **sm**: 表示接收进程消息队列上消息的个数，初值为 0，是控制收发进程同步的信号量

发送原语和接收原语的实现如下：

- **发送原语Send**: 申请一个消息缓冲区，把发送区内容复制到这个缓冲区中；找到接收进程的 PCB，执行互斥操作P(mutex)；把缓冲区挂到接收进程消息队列的尾部，执行V(sm)，即消息数加 1；执行V(mutex)
- **接收原语Receive**: 执行P(sm)查看是否有信件；执行互斥操作P(mutex)，从消息队列中摘下第一个消息，执行V(mutex)；把消息缓冲区内容复制到接收区，释放消息缓冲区



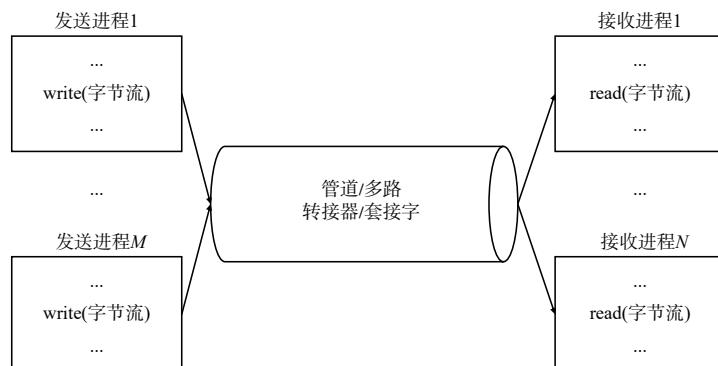
#### 6.1.4 管道和套接字

- 管道是 Unix 和 C 的传统通信方式
- 套接字起源于 Unix BSD 版本，目前已经被 Unix 和 Windows 操作系统广泛采用，并支持 TCP/IP 协议，即支持本机的进程间通信，也支持网络级的进程间通信
- 管道和套接字都是基于信箱的消息传递方式的一种变体，它们与传统的信箱方式等价，区别在于没有预先设定消息的边界。换言之，如果一个进程发送 10 条 100 字节的消息，而另一个进程接收 1000 个字节，那么接收者将一次获得 10 条消息

## 6.2 高级进程通信机制

### 6.2.1 基于流的进程通信

- 多个进程可以使用一个共享的消息缓冲区，这个缓冲区将被组织成一个字节流，而非一个信件流
  - 已有很多操作系统实现了类似的机制，有的称之为管道，有的称之为多路转接器，有的称之为套接字等



- 发送者或称为发送消息的进程向消息缓冲区中写入字节流，接收消息的进程则从消息缓冲区中读取

### 字节流

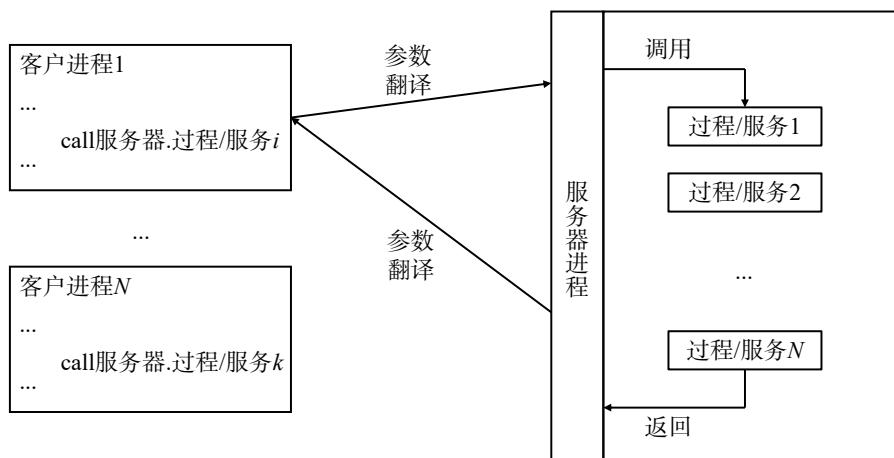
- 可以根据需求任意地读出一个字节流，只要消息缓冲区中有足够的字节数，就可以读出来，如果没有足够的字节数，就去等待消息
- 发送进程在写入字节流时如果消息缓冲区已满，就需要等待该消息缓冲区

## 6.2.2 基于 RPC 的高级通信规约

远程过程调用（RPC）是目前在分布式系统中广泛采用的进程通信方法，它将单机环境下的过程调用拓展到分布式环境中，允许不同计算机上的进程使用简单的过程调用和返回结果的方式进行交互。

- 采用客户/服务器计算模式
- 服务器进程提供一系列过程/服务，供客户进程调用
- 客户进程通过调用服务器进程提供的过程/服务获得服务
- 考虑到客户计算机和服务器计算机的硬件异构型，外部数据表示 XDR 被引入来转换每台计算机的特殊数据格式为标准数据格式

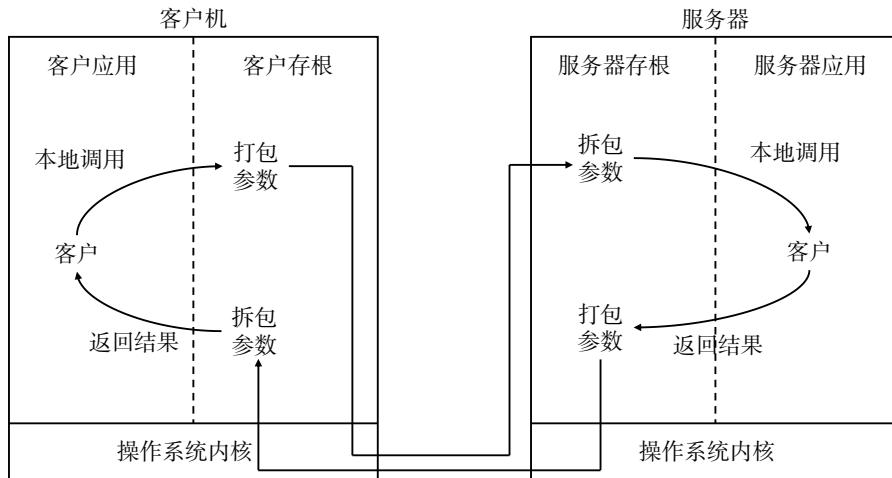
### 基于 RPC/XDR 的高级通信规约



### RPC 的执行步骤：

1. 客户进程以普通方式调用客户存根
2. 客户存根组织 RPC 消息并执行 Send，激活内核程序
3. 内核把消息通过网络发送到远地内核
4. 远地内核把消息送到服务器存根
5. 服务器存根取出消息中参数后调用服务器过程
6. 服务器过程执行完后把结果返回至服务器存根
7. 服务器存根进程将它打包并激活内核程序
8. 服务器内核把消息通过网络发送至客户机内核
9. 客户内核把消息交给客户存根
10. 客户存根从消息中取出结果返回给客户进程

### 11. 客户进程获得控制权并得到了过程调用的结果



## 7 死锁

### 7.1 死锁的产生

#### 7.1.1 死锁的定义

在许多应用中，一个进程需要独占访问不止一个资源，当操作系统允许多个进程并发执行共享系统资源时，可能会出现所有进程被永久阻塞的现象，这时就产生了死锁

一组进程处于死锁状态是指，如果在一个进程集合中，每个进程都在等待只能由该集合中的另一个进程才能引发的事件，则称一组进程或系统此时发生了死锁

- 例如，存在  $n$  个进程  $P_1, P_2, \dots, P_n$
- 进程  $P_i (i = 1, 2, \dots, n)$  因为申请不到资源  $R_j (j = 1, \dots, m)$  而处于等待状态
- 而  $R_j$  又被  $P_{i+1}, (i = 1, \dots, n-1)$  占有， $P_n$  欲申请的资源被  $P_1$  占有
- 显然，此时这  $n$  个进程的等待状态永远不能结束，这  $n$  个进程处于死锁状态

#### 7.1.2 死锁的产生

死锁的产生不仅与系统拥有的资源数量有关，而且与资源分配策略，进程对资源的使用要求以及并发进程的推进顺序有关

##### 7.1.2.1 进程推进顺序不当产生死锁

设系统有打印机、绘图仪各一台，被进程 Q1 和 Q2 共享。两个进程并发执行，按下列次序请求和释放资源：

进程Q1	进程Q2
请求绘图仪	请求打印机
请求打印机	请求绘图仪
使用绘图仪和打印机	使用绘图仪和打印机
释放绘图仪	释放绘图仪
释放打印机	释放打印机

### 7.1.2.2 PV 操作使用不当产生死锁

进程P1	进程P2
...	...
<u>P(s1);</u>	<u>P(s2);</u>
<u>P(s2);</u>	<u>P(s1);</u>
使用r1和r2;	使用r1和r2;
V(s1);	V(s2);
V(s2);	V(s1);

说明：一个系统中有 r1 和 r2 两种资源各 1 个，分别用 s1 和 s2 信号量表示互斥，信号量初始设置为 1

### 7.1.2.3 资源分配不当引起死锁

若系统中有  $m$  个资源被  $n$  个进程共享，每个进程都要求  $K$  个资源，而  $m < nK$  时，即资源数小于进程所要求的总数时，如果分配不当就可能引起死锁

### 7.1.2.4 对临时性资源使用不加限制引起死锁

进程通信使用的信件是一种临时性资源，如果对信件的发送和接收不加限制，可能引起死锁

例如，进程  $P_1$  等待进程  $P_3$  的信件  $S_3$  来到后再向进程  $P_2$  发送信件  $S_1$ ； $P_2$  又要等待  $P_1$  的信件  $S_1$  来到后再向  $P_3$  发送信件  $S_2$ ；而  $P_3$  也要等待  $P_2$  的信件  $S_2$  来到后才能发出信件  $S_3$ 。这种情况下形成了循环等待，产生死锁

## 7.2 死锁的防止

### 7.2.1 死锁产生的条件

系统形成死锁的四个必要条件：

- 互斥条件：系统中存在临界资源，进程应互斥地使用这些资源
- 占有和等待条件：进程请求资源得不到满足而等待时，不释放已占有的资源
- 不剥夺条件：已被占用的资源只能由属主释放，不允许被其它进程剥夺
- 循环等待条件：存在循环等待链，其中，每个进程都在链中等待下一个进程所持有的资源，造成这组进程永远等待
- 前三个条件是死锁存在的必要条件，但不是充分条件，第四个条件是前三个条件同时存在时所产生的结果

### 7.2.2 死锁的防止

只要能破坏上述 4 个必要条件之一，就可防止死锁

- 破坏第一个条件，把独占型资源改造成共享性资源，使资源可同时访问而不是互斥使用。这是一个简单的办法，但对许多资源往往是不能做到的
- 采用剥夺式调度方法可以破坏第三个条件，但剥夺式调度方法目前只适用于对主存资源和处理器资源的分配，而不适用于所有资源

由于上述死锁的防止办法施加于资源的限制条件太严格，会造成资源利用率下降。下面介绍两种比较实用的死锁防止方法，它们能破坏占有和等待条件或循环等待条件

### 7.2.2.1 静态分配策略

静态分配是指一个进程必须在执行前就申请它需要的全部资源，并且直到它需要的资源都得到满足后才开始执行

- 破坏了占有和等待条件
- 该策略实现简单，被许多操作系统采用
- 但该策略严重地降低了资源利用率，因为在每个进程所占有的资源中，有些资源在进程较后的执行时间里才使用，甚至有些资源在例外的情况下才被使用
- 可能导致一个进程占有了一些在其执行周期中使用率很低的资源，而使其他想用这些资源的进程产生等待的结果

### 7.2.2.2 层次分配策略

在层次分配策略下，资源被分成多个层次，一个进程得到某一层的一个资源后，它只能再申请较高一层的资源；当一个进程要释放某层的一个资源时，必须先释放所占用的较高层的资源；当另一个进程获得某一层的一个资源后，它想再申请该层中的另一个资源，必须先释放该层中的已占资源。

- 层次分配策略旨在阻止循环等待条件的出现
- 层次分配比静态分配在实现上要多花一点代价，但它提高了资源使用率
- 然而，如果一个进程使用资源的次序和系统内规定的各层资源的次序不同时，这种提高可能不明显
  - 假如系统中的资源从高到低按序排列为：卡片输入机、行式打印机、卡片输出机、绘图仪和磁带机。若一个进程在执行中较早地使用绘图仪，而仅到快结束时才使用磁带机。但是，若系统规定磁带机所在层次低于绘图仪所在层次，则进程使用绘图仪前就必须先申请到磁带机，该磁带机就在很长一段时间里处于空闲状态直到进程执行到结束前才使用，这无疑是低效率的。

## 7.3 死锁的避免

当不能防止死锁的产生时，如果能掌握并发进程中与每个进程有关的资源申请情况，仍然可以避免死锁的发生

银行家算法是由 Dijkstra 在 1965 年提出的一种资源分配算法：

- 检查申请者对各类资源的最大需求量，如果系统现存的各类资源可以满足它的最大需求量时，就满足当前的申请
- 换言之，仅仅在申请者可能无条件地归还它所申请的全部资源时，才分配资源给它

### 7.3.1 银行家算法的数据结构

一个系统有  $n$  个进程和  $m$  种不同类型的资源，定义包含以下向量和矩阵的数据结构：

- 系统中每类资源总数向量： $\text{Resource} = (R_1, R_2, \dots, R_m)$
- 系统中每类资源当前可用数向量： $\text{Available} = (V_1, V_2, \dots, V_m)$
- 每个进程对各类资源的最大需求矩阵： $\text{Claim}[i, j]$ ，对于  $\text{Claim}[i, j] = k$  表示进程  $P_i$  需要  $R_j$  类资源的最大数目为  $k$  个
- 每个进程已占有各类资源数量矩阵： $\text{Allocation}[i, j]$ ，对于  $\text{Allocation}[i, j] = k$  表示进程  $P_i$  占有  $R_j$  类资源  $k$  个，初始值为 0
- 每个进程尚需要各类资源数量矩阵  $\text{Need}[i, j]$ ，对于  $\text{Need}[i, j] = k$  表示进程  $P_i$  还需要  $R_j$  类资源  $k$  个， $\text{Need}[i, j] = \text{Claim}[i, j] - \text{Allocation}[i, j]$

- 每个进程从当前申请各类资源数量矩阵  $\text{Request}[i, j]$ , 对于  $\text{Request}[i, j] = k$ , 表示进程  $P_i$  当前申请  $R_j$  类资源  $k$  个

在银行家算法中下列关系式确保成立:

- 系统要启动一个新进程工作, 对于资源  $\text{Resource}[i]$  即  $R_i$  的需求满足不等式:

$$R_i \geq \text{Claim}[(n+1), i] + \sum_{k=1}^n \text{Claim}[k, i]$$

也就是应当满足当前系统中所有进程对资源  $R_i$  的最大需求数, 加上启动的新进程的资源最大需求数, 不超过系统拥有的最大资源数时才启动该进程 (进程拒绝启动法)

- 所有资源要么被分配, 要么尚可分配:

$$R_i = V_i + \sum A_{ki}, \quad i = 1, \dots, m, \quad k = 1, \dots, n$$

- 进程申请资源数不能超过系统拥有的资源总数:

$$C_{ki} \leq R_i, \quad i = 1, \dots, m, \quad k = 1, \dots, n$$

- 进程申请任何类资源数不能超过声明的最大资源需求数:

$$A_{ki} \leq C_{ki}, \quad i = 1, \dots, m, \quad k = 1, \dots, n$$

### 7.3.2 银行家算法的思想与实现

系统安全性: 在时刻  $T_0$  系统是安全的, 当且仅当存在一个进程序列  $P_1, \dots, P_n$ , 对进程  $P_k$  满足公式:

$$\text{Need}[k, i] \leq \text{Available}[i](V_i) + \sum_{j=1}^{k-1} \text{Allocation}[j, i] \quad i = 1, \dots, m; \quad k = 1, \dots, n$$

银行家算法的基本思想<sup>1</sup>

1. 系统中的所有进程进入进程集合
2. 在安全状态下系统收到进程的资源请求后, 先把资源试探性分配给它
3. 系统用剩下的可用资源和进程集合中其他进程还要的资源数作比较, 在进程集合中找到剩余资源能满足最大需求量的进程, 从而, 保证这个进程运行完毕并归还全部资源
4. 把这个进程从集合中去掉, 系统的剩余资源更多了, 反复执行上述步骤 (进程退出系统, 资源回收)
5. 最后, 检查进程集合, 若为空表明本次申请可行, 系统处于安全状态, 可实施本次分配; 否则, 有进程执行不完, 系统处于不安全状态, 本次资源分配暂不实施, 让申请进程等待

银行家算法的程序及实现:

1. 如果  $\text{Request}[i, *] \leq \text{Need}[i, *]$ , 转步骤 2; 否则, 进程申请量超过最大需求量, 出错处理
2. 如果  $\text{Request}[i, *] \leq \text{Available}[*]$ , 转步骤 3; 否则, 申请量超过当前系统所拥有的可分配量, 让进程  $P_i$  等待

<sup>1</sup> 安全序列并不是唯一的

3. 系统对  $P_i$  进程请求资源执行试探性分配，执行

$$\begin{aligned} Allocation[i, *] &= Allocation[i, *] + Request[i, *] \\ Available[*] &= Available[*] - Request[i, *] \\ Need[i, *] &= Need[i, *] - Request[i, *] \end{aligned}$$

4. 转向 5 去执行安全性测试算法，如果返回安全状态则承认识试分配，否则抛弃试分配，进程  $P_i$  等待，并执行

$$\begin{aligned} Allocation[i, *] &= Allocation[i, *] - Request[i, *] \\ Available[*] &= Available[*] + Request[i, *] \\ Need[i, *] &= Need[i, *] + Request[i, *] \end{aligned}$$

5. 安全性测试算法

- (1) 定义工作向量  $Work[i]$ 、布尔型标志  $possible$  和进程集合  $rest$
- (2) 执行初始化操作：让全部进程进入  $rest$  集合，并让

$$Work[*] = Available[*], \quad possible = true$$

- (3) 保持  $possible = true$ ，从进程集合  $rest$  中找出满足下列条件的进程  $P_k$

$$Need[k, *] \leq Work[*]$$

- (4) 如果不存在，则转向 (5)；如果找到，则释放进程  $P_k$  所占用的资源，并执行以下操作：

$$Work[*] = Work[*] + Allocation[k, *]$$

把  $P_k$  从进程集合中去掉，即  $rest = rest - \{P_k\}$ ，再转向 (3)

- (5) 置  $possible = false$ ，停止执行本算法
- (6) 最后查看进程集合  $rest$ ，若为空则返回安全标记，否则返回不安全标记

```

1  typedef struct state { //全局数据结构
2      int resource[m];
3      int available[m];
4      int claim[n][m];
5      int allocation[n][m];
6  };
7
8  void resource_allocation() { //资源分配算法
9      if(allocation[i, *] + request[*] > claim[i, *])
10         {error}; //申请量超过最大需求值
11     else {
12         if(request[*] > available[*])
13             {suspend process};
14         else{ //尝试分配, define newstate by:
15             allocation[i, *] = allocation[i, *] + request[*];
16             available[*] = available[*] - request[*];
17         }
18     }
}

```

```

19     if(safe(newstate))
20         {carry out allocation};
21     else{
22         {restore original state};
23         {suspend process};
24     }
25 }

26
27 bool safe(state s) { //安全性测试算法
28     int currentavail[m];
29     set<process> rest;
30     currentavail[*] = available[*];
31     rest = {all process};
32     possible = true;
33     while(possible){ //rest中找一个Pk, 满足以下条件
34         claim[k, *] - allocation[k, *] <= currentavail[*];
35         if(found){
36             currentavail[*] = currentavail[*] + allocation[k, *];
37             rest = rest - {Pk};
38         }else{
39             possible = false;
40         }
41     }
42     return(rest = null);
43 }

```

例：如果系统中共有五个进程和 A、B、C 三类资源，A 类资源共有 10 个，B 类资源共有 5 个，C 类资源共有 7 个，在时刻  $T_0$ ，系统目前资源分配情况如下：

process	Allocation			Claim			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	5	3	3	3	2	7	4	3
$P_1$	2	0	0	3	2	2				1	2	2
$P_2$	3	0	2	9	0	2				6	0	0
$P_3$	2	1	1	2	2	2				0	1	1
$P_4$	0	0	2	4	3	3				4	3	1

可以断言目前系统处于安全状态，因为序列  $\{P_1, P_3, P_4, P_2, P_0\}$  能满足安全性条件

process	Work			Need			Allocation			Work+Allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
$P_1$	3	3	2	1	2	2	2	0	0	5	3	2	true
$P_3$	5	3	2	0	1	1	2	1	1	7	4	3	true
$P_4$	7	4	3	4	3	1	0	0	2	7	4	5	true
$P_2$	7	4	5	6	0	0	3	0	2	10	4	7	true
$P_0$	10	4	7	7	4	3	0	1	0	10	5	7	true

假设  $P_1$  又请求 1 个 A 类资源和 2 个 C 类资源，得到新的状态如下图所示：

$$Request1(1, 0, 2) \leq Need(1, 2, 2) \quad Request1(1, 0, 2) \leq Available(3, 3, 2)$$

process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	1	0	7	4	3	2	3	0
$P_1$	3	0	2	0	2	0			
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	0	0	2	4	3	1			

判定新状态是否安全：执行安全性测试算法，找到一个进程序列  $\{P_1, P_3, P_4, P_0, P_2\}$  能满足安全性条件，所以可正式把资源分配给进程  $P_1$

process	Work			Need			Allocation			Work+Allocation			possible
	A	B	C	A	B	C	A	B	C	A	B	C	
$P_1$	2	3	0	0	2	0	3	0	2	5	3	2	true
$P_3$	5	3	2	0	1	1	2	1	1	7	4	3	true
$P_4$	7	4	3	4	3	1	0	0	2	7	4	5	true
$P_0$	7	4	5	7	4	3	0	1	0	7	5	5	true
$P_2$	7	5	5	6	0	0	3	0	2	10	5	7	true

假设  $P_4$  发起资源请求，按照银行家算法检查，资源不足不予以分配

$$Request4(3, 3, 0) \leq Need(4, 3, 1) \quad Request4(3, 3, 0) > Available(2, 3, 0)$$

假设  $P_0$  发起资源请求，按照银行家算法检查，得到中间结果如下

$$Request0(0, 2, 0) \leq Need(7, 3, 1) \quad Request0(0, 2, 0) \leq Available(2, 3, 0)$$

process	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
$P_0$	0	3	0	7	2	3	2	1	0
$P_1$	3	0	2	0	2	0			
$P_2$	3	0	2	6	0	0			
$P_3$	2	1	1	0	1	1			
$P_4$	0	0	2	4	3	1			

但可以看出系统已处于不安全状态了。

## 7.4 死锁的检测

解决死锁问题的一条途径是死锁的检测和解除

- 这种方法对资源的分配不加任何限制，也不采取死锁的避免措施
- 系统定时地运行一个死锁的检测程序，判断系统内是否已出现死锁，如果检测到系统已发生了死锁，再采取措施解除

死锁的检测可设置两张表格来记录进程使用资源的情况，其中等待资源表记录每个被阻塞进程等待的资源，占用资源表记录每个进程占有的资源

- 任一进程申请资源时，先检查该资源是否为其他进程所占用
- 若资源空闲，则将该资源分配给申请者且登入占用资源表
- 若该资源被其他进程占用，则登入进程等待资源表

资源	占用进程	进程	等待资源
$r_1$	$P_1$	$P_1$	$r_1$
$r_2$	$P_2$	$P_2$	$r_2$
$r_3$	$P_3$	$P_3$	$r_3$
$r_4$	$P_4$	$\dots$	$\dots$
$r_5$	$P_5$		
$\dots$	$\dots$		

死锁检测程序定时检测这两张表，如果有进程  $P_i$  等待资源  $r_k$ ，且  $r_k$  被进程  $P_j$  占用，则说明  $P_i$  和  $P_j$  具有等待占用关系，记为  $W(P_i, P_j)$ 。死锁检测程序反复检测这两张表列出所有的等待占用关系，当出现  $W(P_i, P_j), W(P_j, P_k), \dots, W(P_1, P_m), W(P_m, P_i)$  时，系统中就存在一组循环等待资源的进程  $P_i, P_j, P_k, \dots, P_1, P_m$ ，也就表明系统出现了死锁。

#### 7.4.1 利用等待资源表和占用资源表进行死锁检测

将等待资源表和占用资源表中记录的进程使用和资源等待的情况用一个矩阵  $A$  来表示

$$A[b_{ij}] = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{nn} \end{pmatrix}$$

当  $P_i$  等待被  $P_j$  占用的资源时， $b_{ij}$  的值记为 1；当  $P_i$  和  $P_j$  不存在资源等待占用关系时， $b_{ij}$  的值记为 0

死锁检测程序可用 Warshall 的传递闭包算法来检测系统是否有死锁发生

- Warshall 的传递闭包算法基于状态矩阵  $A$  构成传递闭包  $A * [b_{ij}]$
- $A * [b_{ij}]$  中的每个  $b_{ij}$  是对  $A[b_{ij}]$  执行如下算法得到的

```

1  for(int k = 1; k <= n; k++)
2      for(int i = 1; i <= n; i++)
3          for(int j = 1; j <= n; j++)
4              bij = bij OR (bik AND bkj)

```

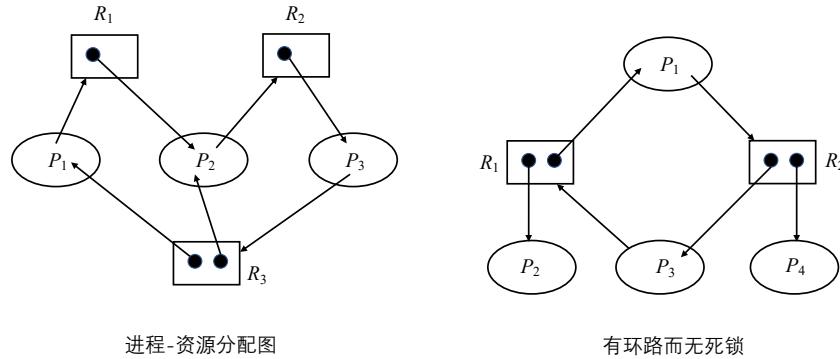
- 其中， $b_{ij}$  表示进程  $P_i$  与进程  $P_j$  存在直接等待关系
- 当进程  $P_i$  等待进程  $P_k$  所占用的资源且进程  $P_k$  等待进程  $P_j$  所占用的资源时， $b_{ik} \wedge b_{kj}$  取值为 1。也就是说，进程  $P_i$  进程  $P_j$  之间具有间接等待关系
- Warshall 传递闭包算法循环检测了矩阵  $A$  中的各个元素，将直接等待资源关系和间接等待资源关系都在传递闭包  $A^*$  中表示出来

- 显然，当  $A^*$  中存在一个  $b_{ii} = 1(i = 1, 2, \dots, n)$  时，就说明存在一组进程处于循环等待资源状态，即系统出现了死锁

### 7.4.2 进程-资源分配图与死锁的检测

进程-资源分配图

- 约定  $P_i \rightarrow R_j$  为请求边，表示进程  $P_i$  申请资源类  $R_j$  中的一个资源得不到满足而处于等待  $R_j$  类资源的状态，该有向边从进程开始指到方框的边缘，表示进程  $P_i$  申请  $R_j$  类中的一个资源。
- 约定  $R_j \rightarrow P_i$  为分配边，表示  $R_j$  类中的一个资源已被进程  $P_i$  占用，由于已把一个具体的资源分给了进程  $P_i$ ，故该有向边从方框内的某个黑圆点出发指向进程。



利用资源分配图检测系统是否处于死锁状态

- 如果进程-资源分配图中无环路，则此时系统没有发生死锁
- 如果进程-资源分配图中有环路，且每个资源类中仅有一个资源，则系统中发生了死锁，此时，环路是系统发生死锁的充要条件，环路中的进程便为死锁进程
- 如果进程-资源分配图中有环路，且涉及的资源类中有多个资源，则环路的存在只是产生死锁的必要条件而不是充分条件
  - 如果能在进程-资源分配图中消去此进程的所有请求边和分配边，这样的进程称为孤立结点。经一系列简化，使所有进程成为孤立结点，则该图是可完全简化的；否则则称该图是不可完全简化的。
  - 系统为死锁状态的充分条件是：当且仅当该状态的进程-资源分配图是不可完全简化的，该充分条件称为死锁定理。

死锁检测算法相应的数据结构：

- $Available[m]$  是长度为  $m$  的向量，说明每类资源中可供分配的资源数目
- $Allocation[n, m]$  是  $n \times m$  矩阵，说明已分配给每个进程的每类资源数目
- $Request[n, m]$  是  $n \times m$  矩阵，说明当前每个进程对每类资源的申请数目
- $Work[m]$  是长度为  $m$  的工作向量
- $finish[n]$  是长度为  $n$  的布尔型工作向量

令  $k = 1, 2, \dots, n$ ，死锁检测算法的步骤如下：

- $Work[*] = Available[*]$
- 如果  $Allocation[k, *] \neq 0$ ，令  $finish[k] = false$ ；否则  $finish[k] = true$

3. 寻找一个  $k$ , 应满足条件

$$(finish[k] == false) \&\& (Request[k, *] \leq Work[*])$$

若找不到这样的  $k$ , 则转向 5

4. 修改  $Work[*] = Work[*] + Allocation[k, *]$ ,  $finish[k] = true$ , 然后转向 3
5. 如果存在  $k (1 \leq k \leq n)$ ,  $finish[k] = false$ , 则系统处于死锁状态, 并且  $finish[k] = false$  的  $P_k$  是处于死锁的进程

#### 7.4.3 死锁的解除

- 结束所有进程的执行, 重新启动操作系统。该方法简单, 但以前的工作全部作废, 损失很大。
- 撤销陷于死锁的所有进程, 解除死锁, 重新启动执行。这种方法的代价也相当大。
- 逐个撤销陷于死锁的进程, 回收其资源重新分派, 直至死锁解除。
- 剥夺陷于死锁的进程占用的资源, 但并不撤销它, 直至死锁解除。可仿照撤销陷于死锁进程的条件来选择剥夺资源的进程。
- 根据系统保存的检查点, 让所有进程回退, 直到足以解除死锁, 这种措施要求系统建立保存检查点、回退及重启机制。
- 当检测到死锁时, 如果存在某些未卷入死锁的进程, 而随着这些进程执行到结束, 有可能释放足够的资源来解除死锁。